TUTTO QUANTO È NECESSARIO CONOSCERE PER REALIZZARE APPLICAZIONI JAVA CHE SI INTERFACCIANO A DATABASE

JAVA & DATABASE

Federico Paparoni

JAVA & DATABASE

di Federico Paparoni

EDIZIONI MASTER

INDICE

INTRODUZIONE

Era una notte buia e tempestosa del 1970, o almeno penso che lo fosse, quando Edgar Frank Codd incominciò a pensare che il modo giusto per gestire dei dati era quello di adottare un modello relazionale, immagazzinando varie informazioni suddivise per tabelle e orchestrabili tramite un linguaggio (SQL). Codd è quello che viene riconosciuto come il padre del database relazionale, perché prima di lui erano presenti altri modelli con i quali venivano rappresentate le informazioni. Il modello relazionale è quello che, col passare degli anni, ha avuto più successo nel campo dell'informatica ed è stato piano piano adottato dalle principali software house (Oracle & friends) come modello di riferimento per i loro prodotti. Il database è una di quelle entità all'interno dei nostri programmi che dobbiamo sempre gestire in qualche maniera. Ogni programma manipola al suo interno dei dati e nella maggior parte dei casi il database è utile per memorizzare e ricercare informazioni necessarie al nostro programma. Specialmente quando dobbiamo realizzare applicazioni commerciali i database offrono una serie di caratteristiche che possono innalzare molto l'affidabilità dei nostri prodotti quindi è importante capire quali siano i diversi modi in cui possiamo "dialogare" con queste entità.

TOOL

All'interno di questo libro vedremo come poter realizzare applicazioni Java che gestiscono i propri dati utilizzando il database. Come database di riferimento, che vedremo quindi in diverse configurazioni, è stato scelto MySQL, ottimo database utilizzato in diversi ambiti e ultimamente acquisito da SUN Microsystems. Vi consiglio quindi di scaricare i seguenti tool per provare i vari esempi che saranno riportati all'interno del libro

- MySQL Community Server 5.0 -
- MySQL GUI Tools: una serie di tool grafici che ci permettono di gestire al meglio il nostro database MySQL -
http://dev.mysql.com/downloads/gui-tools/5.0.html

La natura multipiattaforma di Java ci permette poi di utilizzare diversi database con poche modifiche alla configurazione del nostro codice, quindi altri database che possono essere utilizzati sono ad esempio PostgreSQL (*http://www.postgresql.org/*) o Oracle (*http://www.oracle.com/*).

CAPITOLI

Questo libro cerca di introdurre il lettore in diverse tecnologie che possono essere utilizzate con Java per gestire un database. I capitoli sono così strutturati:

1. Introduzione
2. JDBC: L'API standard Java per collegarsi ad database
3. Hibernate: Famoso tool di ORM (Object Relational Mapping) che permette di ragionare in ottica Object Oriented quando realizziamo le nostre applicazioni
4. iBatis: Altro famoso progetto che si prefigge il compito di essere una sorta di middleware, di "man in the middle" tra la nostra applicazione Object Oriented e il database
5. db4o: Un database Object Oriented, non relazionale, che offre quindi agli sviluppatori un nuovo modo di considerare il database
6. Altre tecnologie

Bisogna sicuramente capire che ognuno di questi capitoli potrebbe richiedere un libro a sé stante, visto che stiamo parlando di tecnologie molto utili ma che, come ogni cosa utile, richiedono talvolta lun-

ghe discussioni ed approfondimenti per essere utilizzati al meglio. In questo libro avremo sicuramente modo di vedere come configurare ed utilizzare questi strumenti, ognuno con peculiarità diverse che possono essere proprio quelle di cui siamo alla ricerca quando dobbiamo realizzare un progetto software. Avremo modo di vedere come alcuni progetti cercano di risolvere l'Impedance Mismatch, ovvero il problema classico che viene generato nel momento in cui proviamo a mappare delle entità Object Oriented in un contesto relazionale.

AUTORE

Federico Paparoni è un Analista Programmatore e per lavoro si occupa principalmente delle piattaforme JavaEE e JavaME. Ha scritto diversi articoli sulla programmazione e gestisce JavaStaff.com, portale per la divulgazione di articoli e novità riguardanti il mondo Java. Per questioni relative al libro può essere contattato all'indirizzo email *federico.paparoni@javastaff.com*

JDBC

La SUN fin dalle prime versioni di Java ha portato avanti un API che permette alle applicazioni scritte in questo linguaggio di avere un livello di astrazione uniforme verso i database: JDBC. Questa API, il cui nome in realtà non è secondo i progettisti SUN un acronimo di Java Database Connectivity, è stata inclusa nel JDK a partire dalla versione 1.1, caratterizzando fin dall'inizio questo linguaggio come multipiattaforma anche per quanto riguardava la connessione al database. Infatti, attraverso JDBC, vengono dati allo sviluppatore principalmente 3 strumenti che possono essere utilizzati allo stesso modo con tutti i database

- Una connessione verso il database in questione
- La possibilità di inviare dei comandi SQL
- Ricevere una risposta dal database che può contenere dei dati

Basandosi sulla generalizzazione di questi concetti e su una serie di driver differenti per ogni diverso database, JDBC ci permette dalla nostra applicazione Java di gestire completamente tabelle, query, sequenze e quant'altro relativo al mondo dei database. Questa omogeneità di API viene resa possibile attraverso dei driver che da una parte implementano le interfacce definite all'interno di JDBC. Dall'altra gestiscono la connessione verso il database, trasformando le chiamate JDBC in chiamate verso il database utilizzando il rispettivo protocollo. In questo modo abbiamo la possibilità di avere dal punto di vista della programmazione una sola interfaccia nota da gestire, quella JDBC e possiamo cambiare il database semplicemente cambiando il driver che leghiamo alla nostra applicazione. I driver JDBC possono essere di diverse tipologie, come quelle che vengono riportate qui di seguito:

- Tipo 1. Bridge JDBC-ODBC: Si tratta di un driver da utilizzare nel caso in cui non sia stata realizzata una vera e propria implementazio-

ne JDBC verso un certo database e quindi si utilizza questo "driver-ponte" che converte le chiamate JDBC in ODBC (Open Database Connectivity), una tecnologia simile ma più vecchia e realizzata solitamente utilizzando linguaggi come C/C++

- Tipo 2. API Native: In questo caso le chiamate JDBC non vengono tradotte in ODBC ma vengono utilizzate delle API native realizzate appositamente per il database. Il tipo 2 è più prestante del tipo 1, anche se ha come svantaggio il fatto di dover utilizzare per forza il client proprietario del database che fornisce queste API native.
- Tipo 3. Driver di rete: Questi driver, rispetto ai precedenti, hanno il vantaggio di essere scritti totalmente in Java e di utilizzare un middleware per effettuare le chiamate, che nel caso di una LAN non congestionata potrebbe essere un vantaggio.
- Tipo 4. Driver nativo in Java: L'ultima tipologia di driver è quello che nella maggior parte dei casi risulta essere il migliore perché è realizzato totalmente in Java e trasforma le chiamate JDBC direttamente nel protocollo del database, quindi permette un collegamento diretto senza bisogno di middleware o client installati.

Attualmente JDBC è arrivato alla versione 4.0, introducendo pian piano sempre delle interessanti novità. A partire dalla versione 3.0 tutto le decisioni e lo sviluppo relativo a JDBC viene effettuato tramite JCP (Java Community Process, per maggiori informazioni *http://jcp.org/*). Le specifiche realizzate in questo modo sono quelle relative a JDBC 3.0 (JSR 54), il Rowset (JSR 114) e JDBC 4.0 (JSR 221). Tendenzialmente, per quanto riguarda i driver, sono direttamente coloro che realizzano il database a fornire un'implementazione JDBC compliant. In questo caso bisogna vedere a quale versione di JDBC aderiscono questi driver per sapere le diverse opzioni che possono essere utilizzate all'interno dei nostri programmi. Inoltre driver diversi potrebbero anche avere prestazioni differenti. Per esempio, insieme alla versione 8.2 di PostgreSQL vengono distribuite 4 differenti versioni di driver da utilizzare con la nostra applicazione. Per entrare nel vivo delle API JDBC e incominciare a capi-

re qualcosa, dobbiamo prima di tutto vedere come è possibile ottenere una connessione verso il nostro database.

DRIVERMANAGER

JDBC ci permette di avere un'unica API per gestire nello stesso modo diverse tipologie di database. Il nostro programma deve comunque specificare qualche informazione per comunicare a JDBC quale è il database che vogliamo utilizzare, dove si trova, come è possibile accedere. Fornendo queste informazioni potremo poi ottenere un oggetto *java.sql.Connection*, quello che ci permette di avere/settare informazioni sulla connessione verso il database ed inviare successivamente dei veri e propri comandi SQL. Uno dei modi possibili per ottenere questo oggetto *Connection* è quello di utilizzare *DriverManager*, una classe che attraverso dei metodi *getConnection()* permette di ottenere questa connessione al database. Chiaramente questa classe deve avere l'informazione relativa anche al driver che vogliamo utilizzare. Per fare ciò dobbiamo caricare la classe relativa presente nel driver che implementa l'interfaccia *java.sql.Driver*, come riportato di seguito

```
Class.forName("org.postgresql.Driver");
```

Questa, a sua volta, effettua la registrazione del driver sul DriverManager utilizzando il metodo statico *registerDriver()*. Quindi, una volta che abbiamo scritto il precedente comando noi abbiamo caricato il driver e dobbiamo soltanto preoccuparci di fornire al DriverManager le altre informazioni necessarie per ottenere la connessione. Qui di seguito è possibile vedere un primo esempio dove viene ottenuta una Connection che punta al nostro database.

```
package it.ioprogrammo.librodb;

import java.sql.Connection;
```

```
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.DatabaseMetaData;

public class TestConnection {

    public static void main(String a[]) throws Exception {
        Class.forName("org.postgresql.Driver");
        String url = "jdbc:postgresql:libro";
        Connection connection = DriverManager.getConnection(url,
                                                    "postgres", "postgres");
        DatabaseMetaData metadata=connection.getMetaData();
        String driver=metadata.getDriverName()+"
                                        "+metadata.getDriverVersion();
        String urlconn=metadata.getURL();
        String funzioni=metadata.getStringFunctions();
        boolean supportaSP=metadata.supportsStoredProcedures();
        boolean supportaT=metadata.supportsTransactions();
        connection.close();
        System.out.println("Driver: "+driver);
        System.out.println("URL: "+urlconn);
        System.out.println("Funzioni presenti: "+funzioni);
        System.out.println("Supporta Stored Procedures: "+supportaSP);
        System.out.println("Supporta transazioni: "+supportaT);
    }

}
```

La connessione viene ottenuta fornendo url, username e password. In questo caso stiamo parlando di URL JDBC, ovvero stringhe che identificano il nostro database utilizzando la struttura seguente

```
jdbc:<subprotocol>:<subname>
```

e comunicando quindi al nostro DriverManager quale driver utilizzare, perché ogni driver che si registra gestisce diverse tipologie di url. Quindi quando noi passiamo al DriverManager un certo url, questo interroga la lista dei driver che ha registrato e vede quale utilizzare per ottenere una connessione. All'interno dell'URL JDBC è possibile anche inserire l'host di destinazione, la porta e lo schema da utilizzare all'interno del database. Per quanto riguarda questa informazione potete rivolgervi alla documentazione che viene fornita insieme al driver che dovete utilizzare (chiaramente variano rispetto al database utilizzato e al driver JDBC). Ottenuta una *Connection* la utilizziamo subito per avere qualche informazione sul nostro database. Per fare ciò abbiamo bisogno di DatabaseMetadata che ci permette di sapere molte cose, ad esempio le varie feature che vengono supportate. Di seguito trovate l'output del programma precedente

```
Driver: PostgreSQL Native Driver PostgreSQL 8.2 JDBC2 with NO SSL
                                                        (build 505)
URL: jdbc:postgresql:libro
Funzioni presenti: ascii,char,concat,lcase,left,length,
                      ltrim,repeat,rtrim,space,substring,ucase,replace
Supporta Stored Procedures: true
Supporta transazioni: true
```

In questo caso abbiamo a che fare con PostgreSQL 8.2, altro famoso database opensource, il quale supporta sia le transazioni che le stored procedure. Se avessimo voluto utilizzare MySQL, database che verrà utilizzato in questo libro anche per altri esempi, avremmo dovuto sicuramente cambiare la stringa di connessione JDBC nella seguente

```
jdbc:mysql://localhost:3306/librodb
```

Reperite le informazioni, passiamo, quindi, alla chiusura della connessione, cosa che deve essere sempre fatta come una buona regola di

programmazione, sia perché libera memoria sia perché si liberano connessioni verso il database senza lasciarle appese

```
connection.close();
```

La classe *Connection* mette a disposizione anche altri interessanti metodi che possiamo utilizzare per gestire la connessione con il database. Il metodo *setTransactionIsolation()* permette, ad esempio, di definire il livello di isolamento relativo alle transazioni che possiamo gestire attraverso JDBC. I livelli che possono essere definiti sono i classici quattro relativi al mondo dei database

- TRANSACTION_READ_COMMITTED: Vengono prevenute le letture sporche. Possono comunque essere presenti delle letture fantasma o letture non ripetibili
- TRANSACTION_READ_UNCOMMITTED: Possono essere presenti letture sporche, letture fantasma e letture non ripetibili
- TRANSACTION_REPEATABLE_READ: Limita i possibili problemi lasciando le letture fantasma
- TRANSACTION_SERIALIZABLE: Elimina tutti e 3 i problemi relativi alle letture

Oltre a questo possiamo disabilitare l'autocommit del nostro database, gestendo una vera e propria transaziona relativa ai vari comandi che vogliamo inviare al database.

```
connection.setAutoCommit(false);
...
...
try {
      //QUI EFFETTUIAMO OPERAZIONI DI CUI
//NON VIENE FATTO IL COMMIT IMMEDIATAMENTE
...
```

```
...
//ALLE FINE EFFETTUIAMO IL COMMIT

connection.commit();
}
catch(SQLException e) {
      connection.rollback();
}
```

Come ogni altra API java, anche con JDBC abbiamo la classica gestione dei problemi con le Exception. Una delle eccezioni classiche che avremo modo di vedere utilizzando JDBC è *SQLException*, che contiene al suo interno dei messaggi informativi inviati dal database sottostante, il quale ci spiega per quale motivo la nostra applicazione ha riportato un errore. Nel caso in cui i comandi che stiamo inviando verso il database provochino un SQLException allora il nostro blocco try-catch ci permette di non effettuare nessuna commit ma di effettuare il rollback di tutte le operazioni della nostra transazione. In aggiusta a questo meccanismo di commit/rollback, JDBC ci permette di salvare diversi step all'interno della transazione che stiamo gestendo. Questo è possibile utilizzando il metodo *setSavePoint()* di *Connection* che ci restituisce un oggetto *SavePoint* relativo al momento della transazione in cui siamo. Successivamente possiamo ripristinare la nostra transazione tornando direttamente al punto che abbiamo salvato, utilizzando il metodo *rollback()* di *Connection* che prende come parametro un *SavePoint*.

DATASOURCE

Il DriverManager non è l'unico modo in cui possiamo ottenere una Connection JDBC per dialogare con il database. L'interfaccia *DataSource*, definita nel package *javax.sql*, permette di ottenere una connessione tramite JNDI (*Java Naming and Directory*), definendo ovvero la connessione al di fuori della nostra applicazione e permettendoci, quindi, di

non dover includere nelle nostre applicazioni configurazioni che possono poi cambiare. Oltre a questo interessante vantaggio rispetto al DriverManager, attraverso il DataSource possiamo utilizzare in maniera trasparente per il nostro programma le seguenti feature:

- **Connection pooling**: La connessione che otteniamo sarà una di quelle presenti in un pool di connessioni, in questo modo i tempi di attivazione verso il database sono ridotti
- **Transazioni distribuite**: La connessione in questo caso può partecipare ad una transazione distribuita

Vediamo ora come poter utilizzare un DataSource e come possiamo ottenere da esso una Connection. La configurazione che viene ora riportata riguarda Apache Tomcat, Servlet/JSP container dove possiamo comodamente configurare dei DataSource per le nostre applicazioni

```
<Resource name="jdbc/TestDB" auth="Container"
type="javax.sql.DataSource"
            maxActive="100" maxIdle="30" maxWait="10000"
            username="librodb" password=" librodb"
                        driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/librodb?autoReconnect=true"/>
```

Questa configurazione deve essere presente all'interno del file *context.xml* oppure all'interno di *server.xml*, file di configurazione presenti in Tomcat. In questo modo abbiamo configurato un DataSource che può essere richiamato utilizzando il nome jdbc/TestDB. Come potete vedere abbiamo dovuto inserire nella configurazione anche l'url JDBC, username e password, in modo tale che quando utilizzeremo questo DataSource dal nostro programma dovremo semplicemente richiamarlo attraverso JNDI. Questa configurazione possiamo farla manualmente o gestirla direttamente dalla console di amministrazione di Tomcat, come viene riportato nell'immagine.

Figura 1.1: Configurazione DataSource da Tomcat

Per utilizzare poi questo DataSource dalla nostra applicazione web, dobbiamo inserire la risorsa nel file *web.xml*

```
<resource-ref>
    <description>DB Connection</description>
    <res-ref-name>jdbc/TestDB</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

Una volta che abbiamo queste informazioni configurate possiamo tranquillamente richiamare il DataSource dal nostro programma ed ottenere una Connection per il nostro database. Vediamo quindi una semplice Servlet, dove abbiamo ridefinito il metodo *init()* per vedere se all'avvio riesce a collegarsi al database tramite il DataSource da noi definito:

```
package it.ioprogrammo.librodb;
import java.io.*;
```

```
import java.net.*;
import java.sql.Connection;
import java.sql.SQLException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.sql.DataSource;
public class ProvaDataSource extends HttpServlet {

    public void init() throws ServletException {
        Context initContext;
        try {
            initContext = new InitialContext();
            Context envContext  =
                                    (Context)initContext.lookup("java:/comp/env");
            DataSource ds = (DataSource)envContext.lookup("jdbc/TestDB");
            Connection conn = ds.getConnection();
          System.out.println(conn.getMetaData().getDatabaseProductName());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

I DataSource possono essere definiti in maniera analoga anche su altri container diversi da Tomcat. Ad esempio, se dobbiamo utilizzare BEA WebLogic per un nostro progetto, possiamo tranquillamente definire tutta la configurazione riguardante i DataSource e i pool di connessioni ad essi associati, direttamente dall'interfaccia di amministrazione, come potete vedere dalla figura seguente.
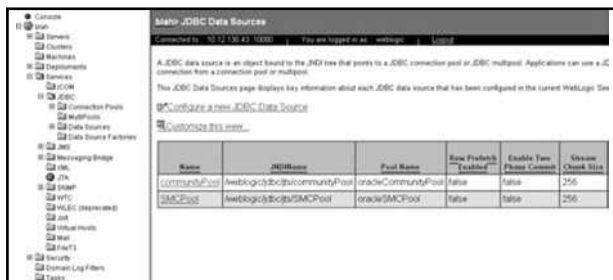
Figura 1.2: Configurazione DataSource da BEA WebLogic

STATEMENT E RESULTSET

Abbiamo visto come poter ottenere la connessione tramite le API JDBC in diversi modi, ora dobbiamo vedere come poter inviare dei comandi al database e gestire i risultati. La classe che ci permette di inviare un comando al database è *Statement*, una volta che abbiamo una *Connection* si può ottenere nel seguente modo:

```
Statement statement = connection.createStatement();
```

Lo Statement può essere usato in 4 diversi modi per inviare delle istruzioni SQL al nostro database, utilizzando 4 diversi metodi che offre questa classe:

- **executeQuery()**: Viene eseguito il comando SQL che è fornito come argomento al metodo. Questo metodo permette di avere come risposta un ResultSet, che è l'insieme dei risultati derivanti dalla nostra query
- **executeUpdate()**: Attraverso questo metodo possiamo inviare al database quei comandi SQL che non prevedono un insieme di risultati, ma quelli che riguardano la gestione del database (SQL DDL). Questo metodo si utilizza per comandi sql come UPDATE, INSERT, DELETE e ritorna il numero di righe che sono state interessate dal no-

stro comando

- **execute()**: Questo metodo viene utilizzato in quei particolari (e rari) casi in cui la risposta al nostro comando SQL può contenere diverse risposte, ovvero diversi oggetti ResultSet.
- **executeBatch()**: E' possibile inserire nel nostro Statement una serie di comandi SQL (INSERT o UPDATE di solito) che possono essere eseguite in sequenza da JDBC quando richiamiamo questo metodo, che ritorna un array di interi dove vengono specificate per ogni comando SQL le righe che sono state modificate tramite esso.

Se vogliamo quindi creare una nostra tabella tramite JDBC, e incominciare ad inserire delle righe, dobbiamo utilizzare il metodo *executeUpdate()*, come potete vedere nell'esempio seguente:

```
package it.ioprogrammo.librodb;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.logging.Level;
import java.util.logging.Logger;

public class TestExecuteUpdate {
    public static void main(String a[]) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            String url = "jdbc:mysql://localhost:3306/libro";
           Connection connection = DriverManager.getConnection(url, "root",
                                                                   "mysql");
            Statement statement = connection.createStatement();
            String creazioneTabella="CREATE TABLE `libro`.`UTENTI` (" +
                            "`ID` INTEGER UNSIGNED NOT NULL
```

```
                                              AUTO_INCREMENT," +
                         "`NOME` VARCHAR(45) NOT NULL," +
                         "`COGNOME` VARCHAR(45) NOT NULL," +
                         "`TELEFONO` VARCHAR(45) NOT NULL," +
                         "`EMAIL` VARCHAR(45) NOT NULL," +
                         "PRIMARY KEY (`ID`))";
      int risultato=statement.executeUpdate(creazioneTabella);
      Logger.getLogger("TestExecuteUpdate").log(Level.INFO, "Risultato
                                              CREATE :"+risultato);
      String inserimentoUtente="INSERT INTO
                  `libro`.`UTENTI`(NOME,COGNOME,TELEFONO,EMAIL) " +
"VALUES('Federico','Paparoni','1234567890','federico.paparoni@javastaff.
                                                          com')";
      risultato=statement.executeUpdate(inserimentoUtente);
      Logger.getLogger("TestExecuteUpdate").log(Level.INFO, "Risultato
                                              INSERT :"+risultato);
       rs.close();
       connection.close();
    } catch (SQLException ex) {
      Logger.getLogger("TestExecuteUpdate").log(Level.SEVERE, null, ex);
    } catch (ClassNotFoundException ex) {
      Logger.getLogger("TestExecuteUpdate").log(Level.SEVERE, null, ex);
    }

  }
}
```

In questo breve esempio abbiamo creato una connessione verso il nostro database MySQL e a partire dalla connessione abbiamo inizializzato uno Statement. Dopo di ciò abbiamo inviato due diversi comandi, uno per la creazione della tabella ed il successivo per l'inserimento di una entry. Entrambe le operazioni sono state realizzate utilizzando il metodo *executeUpdate()* di *Statement*, passando come argomento una

stringa con il comando SQL. Passiamo ora al metodo più importante di *Statement*, *executeQuery()*, che ci permette di avere come risultato un insieme di entry relative ad una o più tabelle. *ResultSet* è la classe che mappa questo concetto, permettendoci di avere a disposizioni tutti i valori restituiti dalla query sotto forma di oggetto Java. Quando richiamiamo il metodo *executeQuery()* ed otteniamo un oggetto *ResultSet*, avremo la risposta alla nostra query inserita in questo oggetto, organizzata secondo le righe della selezione che abbiamo fatto. Per questo motivo possiamo navigare tutta la risposta riga dopo riga, utilizzando l'operatore *next()* di *ResultSet* che ci permette di andare avanti nella risposta restituita fino al suo completamento:

```
Statement statement = connection.createStatement();
ResultSet rs = stmt.executeQuery("SELECT ID, NOME, COGNOME FROM
                                                        UTENTI");
while (rs.next()) {
      int id = rs.getInt("ID");
      String nome = rs.getString("NOME");
      String cognome = rs.getFloat("COGNOME");
      System.out.println("UTENTE #" + id + " : " + nome + " " + cognome);
}
```

Come potete osservare, il ciclo *while* continua fino a quando è presente una riga nel nostro *ResultSet*, cioè fino a quando non lo abbiamo visitato tutto in una determinata direzione. All'interno del ciclo andiamo ad estrapolare i dati utilizzando i metodi *getXXX* di *ResultSet*. Questi metodi vengono utilizzati per effettuare il giusto cast tra il tipo di dato presente sul database e quello che noi dobbiamo importare nella nostra applicazione Java. Come parametro può essere passato il nome della colonna che ci interessa o la sua posizione relativa all'interno della query (ad esempio 2 se la colonna è la seconda nella SELECT). Quando andiamo ad utilizzare questi metodi dobbiamo sapere bene quale tipo di dato stiamo trattando, altrimenti possiamo sollevare delle eccezioni. Result-

Set è un oggetto che ha acquisito con le varie versioni di JDBC diverse feature interessanti, che andiamo ora ad analizzare. Come abbiamo già detto, dobbiamo analizzare le informazioni contenute nel ResultSet procedendo riga per riga, grazie al metodo *next()* che ci permette di saltare alla riga successiva. Proprio su questa forma di gestione dei dati (a righe con avanzamento tramite un cursore) che otteniamo come risposta alla nostra query possiamo avere a che fare con diverse tipologie di *ResultSet:*

- **TYPE_FORWARD_ONLY**: E' possibile analizzare i dati restituiti andando in una sola direzione, cioè muovendosi riga per riga senza poter tornare indietro
- **TYPE_SCROLL_INSENSITIVE**: In questo caso possiamo andare avanti ed indietro nell'analisi del ResultSet, utilizzando alcuni metodi che ci permettono appunto di non essere obbligati ad un analisi dei dati in una sola direzione
- **TYPE_SCROLL_SENSITIVE**: Come il precedente ma inoltre c'è la possibilità che se dal momento in cui è stata effettuata la query i dati del database presenti nel ResultSet sono cambiati, allora questi cambiamenti si ripercuoteranno anche all'interno del ResultSet

Con l'ultima tipologia abbiamo quindi anche la possibilità di avere i dati aggiornati all'interno del nostro ResultSet. Un'altra interessante possibilità offerta dal ResultSet è quella che aggiornando i dati al suo interno venga aggiornato anche il database. Questa feature porta due ulteriori tipologie di *ResultSet:*

- **CONCUR_READ_ONLY**: Il ResultSet è di sola lettura
- **CONCUR_UPDATABLE**: Il ResultSet può essere aggiornato e di conseguenza viene aggiornato il database. In questo caso però abbiamo un livello di concorrenza più basso, permettendo l'accesso al database ad un numero minore di utenti per preservare la coerenza dei dati

Queste caratteristiche che abbiamo visto, ed altre che non avremo modo di vedere, sono spesso dipendenti dall'implementazione JDBC che stiamo utilizzando. Può succedere, ad esempio che il driver per un certo database non supporti la possibilità di aggiornare il ResultSet. Se il driver supporta queste feature (incluse in JDBC 2.0) allora possiamo creare i nostri ResultSet utilizzando i seguenti metodi

```
//In questo caso verrà poi creato un ResultSet
//di tipo TYPE_FORWARD_ONLY e CONCUR_READ_ONLY
Statement statement = connection.createStatement();
...
...
//Ora invece avremo un ResultSet
//di tipo TYPE_SCROLL_SENSITIVE e CONCUR_UPDATABLE
Statement statement =
        connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

Per essere certi di poter utilizzare queste e magari altre feature presenti in JDBC è sempre il caso di controllare cosa viene detto dal fornitore del driver e soprattutto esaminare la feature che desideriamo attraverso l'oggetto *DatabaseMetaData* che abbiamo già visto. In questo caso per sapere se il nostro driver supporta queste due feature, possiamo richiamare i metodi *supportsResultSetType()* e *supportsResultSetConcurrency()*. Vediamo ora un esempio dove utilizziamo il ResultSet per aggiornare un campo attraverso la feature *CONCUR_UPDATABLE:*

```
Statement statement =
connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.
                                                    CONCUR_UPDATABLE);
ResultSet rs=statement.executeQuery("SELECT ID,TELEFONO,EMAIL FROM
      UTENTI WHERE NOME='FEDERICO' AND COGNOME='PAPARONI'");
while(rs.next()) {
```

```
//ORA COMBINIAMO QUALCHE PASTICCIO INVERTENDO
     //I DUE VALORI E AGGIORNANDO LA RIGA SUL DB
     String email = rs.getString("EMAIL");
     String telefono = rs.getString("TELEFONO");
     rs.updateString("EMAIL", telefono);
     rs.updateString("TELEFONO", email);
     rs.updateRow();
}
```

Per effettuare l'aggiornamento di una riga abbiamo prima cambiato i vari campi e poi richiamato il metodo *updateRow()*, il quale effettua il vero e proprio UPDATE. Come potete vedere abbiamo creato lo Statement inserendo la tipologia di ResultSet *CONCUR_UPDATABLE*. Oltre a questo abbiamo dovuto inserire nella nostra SELECT la chiave primaria della tabella ed abbiamo selezionato una sola tabella. Questo perché, se avessimo fatto in maniera differente, saremmo andati contro le regole per il ResultSet aggiornabile definite dalle specifica JDBC e subito il driver MySQL avrebbe segnalato il problema

```
com.mysql.jdbc.NotUpdatable: Result Set not updatable.This result set must
            come from a statement that was created with a result set type of
ResultSet.CONCUR_UPDATABLE, the query must select only one table, and
            must select all primary keys from that table. See the JDBC 2.1 API
                                    Specification, section 5.6 for more details.
      at com.mysql.jdbc.UpdatableResultSet.generate
                                    Statements(UpdatableResultSet.java:558)
      at com.mysql.jdbc.UpdatableResultSet.sync
                                    Update(UpdatableResultSet.java:1344)
      at com.mysql.jdbc.UpdatableResultSet.
                                    updateString(UpdatableResultSet.java:2301)
      at com.mysql.jdbc.UpdatableResultSet.
                                    updateString(UpdatableResultSet.java:2339)
      at it.ioprogrammo.librodb.TestResultSet.main(TestResultSet.java:25)
```

LA SOTTOCLASSE PREPAREDSTATEMENT

Ora che abbiamo visto come utilizzare Statement nelle nostre applicazioni possiamo vedere *PreparedStatement*, che è una sua sottoclasse con alcune funzionalità aggiuntive. Statement permette di passare una vera e propria stringa SQL ai suoi metodi *executeQuery* ed *executeUpdate* per inviare il comando di cui abbiamo bisogno. *PreparedStatement* permette di definire un comando parametrizzandolo, ovvero inserendo delle variabili che verranno poi gestite dal programma a tempo di esecuzione. Praticamente quando dobbiamo fare una SELECT, con Statement passiamo come argomento tutta la stringa SQL

```
SELECT ID,TELEFONO,EMAIL FROM UTENTI WHERE NOME='FEDERICO'
                                        AND COGNOME='PAPARONI'
```

In questo caso, invece, possiamo parametrizzare il comando, in modo tale che possa essere utilizzato anche con altri valori

```
SELECT ID,TELEFONO,EMAIL FROM UTENTI WHERE NOME=? AND
                                                COGNOME=?
```

PreparedStatement offre quindi dei metodi che permettono di settare i vari campi, come riportato nell'esempio seguente:

```
PreparedStatement preparedStatement = connection.prepareStatement
      ("SELECT ID,TELEFONO,EMAIL FROM UTENTI WHERE NOME=? AND
                                                COGNOME=?");
preparedStatement.setString(1, "FEDERICO");
preparedStatement.setString(2, "PAPARONI");
ResultSet rs=preparedStatement.executeQuery();
while(rs.next()) {
String email = rs.getString("EMAIL");
    String telefono = rs.getString("TELEFONO");
```

```
    System.out.println("Dati: "+email+" "+telefono);
}
rs.close();
connection.close();
```

I *PreparedStatement* aumentano l'efficienza del nostro codice nel caso in cui vengano utilizzati su query che vengono effettuate frequentemente. Un esempio classico di utilizzo è quello relativo all'inserimento dei dati, dove parametrizziamo tutta la query e possiamo quindi "wrappare" il metodo di inserimento in un nostro metodo/classe che ci permette di gestire gli inserimenti passando i giusti parametri.

GLI OGGETTI ROWSET

Un ulteriore strumento che possiamo utilizzare di JDBC sono i *RowSet*. Questi oggetti, che estendono *ResultSet*, permettono una gestione migliore dei dati recuperati da un database. Il *RowSet* implementa il modello JavaBeans, quindi ha una gestione delle varie proprietà con i relativi metodi get/set e un modello ad eventi che permette di agganciare a questo oggetto un listener, che viene notificato quando si verifica uno di questi 3 casi

- Il cursore di lettura del RowSet viene spostato
- C'è un inserimento, update o cancellazione riguardante una delle sue righe
- Vengono cambiati tutti i dati relativi al RowSet

Esistono diverse implementazioni di RowSet, che si differenziano prima di tutto per quanto riguarda la connessione. Un RowSet, infatti, può essere connesso o meno, ovvero può fornire il risultato di un determinato comando SQL stando connesso o meno al database. Oltre a questo c'è da dire che i RowSet possono essere associati anche a fonti dati differenti da un database, quindi il loro utilizzo potrebbe essere interes-

sante anche in altri ambiti. Vediamo ora quali sono i principali RowSet che vengono forniti da SUN:

- **JdbcRowSet**: è l'unico RowSet che rimane connesso al database. Molto utile per la gestione dei dati attraverso il modello a eventi e soprattutto per la possibilità di avere un ResultSet aggiornabile e che permette lo scorrimento, visto che ogni implementazione di RowSet lo deve permettere
- **CachedRowSet**: è la prima implementazione dei RowSet disconnessi, quelli che praticamente tengono tutti i dati in memoria e che si collegano al database solo quando è strettamente necessario. Questa implementazione permette anche di serializzare l'oggetto per condividerlo in un ambiente distribuito
- **WebRowSet**: estende *CachedRowSet* e permette, inoltre, di convertire tutto il contenuto del *RowSet* in un documento XML
- **JoinRowSet**: estende *WebRowSet* ed aggiunge la possibilità di effettuare il JOIN SQL tra due diversi *RowSet* senza avere la necessità di riconnettersi al database
- **FilteredRowSet**: estende *WebRowSet* ed aggiunge la possibilità di raffinare il contenuto del *RowSet* in base a determinati filtri

SUN ha distribuito oltre alle interfacce di questi *RowSet* anche una sua implementazione che possiamo utilizzare ed estendere a nostro piacimento. Vediamo ora l'utilizzo di uno dei *RowSet* più interessanti, *WebRowSet:*

```
package it.ioprogrammo.librodb;

import com.sun.rowset.WebRowSetImpl;
import java.io.FileWriter;
import java.io.IOException;
import java.sql.SQLException;
import java.util.logging.Level;
```

```
import java.util.logging.Logger;
public class TestWebRowSet {
    public static void main(String a[]) {
        {
            FileWriter writer = null;
            try {
                Class.forName("com.mysql.jdbc.Driver");
                String url = "jdbc:mysql://localhost:3306/libro";
                WebRowSetImpl webRowSet = new WebRowSetImpl();
                webRowSet.setCommand("SELECT NOME,COGNOME FROM
                                                        UTENTI");
                webRowSet.setUrl("jdbc:mysql://localhost:3306/libro");
                webRowSet.setUsername("root");
                webRowSet.setPassword("mysql");
                webRowSet.execute();
                writer = new FileWriter("test.xml");
                webRowSet.writeXml(writer);
                webRowSet.close();
            } catch (IOException ex) {
Logger.getLogger(TestWebRowSet.class.getName()).
                                        log(Level.SEVERE, null, ex);
            } catch (SQLException ex) {
Logger.getLogger(TestWebRowSet.class.getName()).
                                        log(Level.SEVERE, null, ex);
            } catch (ClassNotFoundException ex) {
Logger.getLogger(TestWebRowSet.class.getName()).
                                        log(Level.SEVERE, null, ex);
            }
            finally {
                try {
                    writer.close();
                } catch (IOException ex) {
Logger.getLogger(TestWebRowSet.class.getName()).
```

```
                                                    log(Level.SEVERE, null, ex);
                }
            }
        }

    }
}
```

In questo caso non abbiamo avuto bisogno di creare una connessione come nei casi precedenti, ma è direttamente il *WebRowSet* che la gestisce. Sono stato settati tutti i parametri necessari attraverso dei metodi set ed infine abbiamo trasformato il contenuto del *WebRowSet* in un file xml, salvandolo su filesystem:

```
<?xml version="1.0"?>
<webRowSet xmlns="http://java.sun.com/xml/ns/jdbc"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/jdbc
                        http://java.sun.com/xml/ns/jdbc/webrowset.xsd">
  <properties>
    <command>SELECT NOME,COGNOME FROM UTENTI</command>
    <concurrency>1008</concurrency>
    <datasource><null/></datasource>
    <escape-processing>true</escape-processing>
    <fetch-direction>1000</fetch-direction>
    <fetch-size>0</fetch-size>
    <isolation-level>2</isolation-level>
    <key-columns>
    </key-columns>
    <map>
    </map>
    <max-field-size>0</max-field-size>
    <max-rows>0</max-rows>
```

```
    <query-timeout>0</query-timeout>
    <read-only>true</read-only>
    <rowset-type>ResultSet.TYPE_SCROLL_INSENSITIVE</rowset-type>
    <show-deleted>false</show-deleted>
    <table-name>UTENTI</table-name>
    <url>jdbc:mysql://localhost:3306/libro</url>
    <sync-provider>
      <sync-provider-name>com.sun.rowset.providers.RIOptimisticProvider
                                         </sync-provider-name>
      <sync-provider-vendor>Sun Microsystems Inc.</sync-provider-vendor>
      <sync-provider-version>1.0</sync-provider-version>
      <sync-provider-grade>2</sync-provider-grade>
      <data-source-lock>1</data-source-lock>
    </sync-provider>
  </properties>
  <metadata>
    <column-count>2</column-count>
    <column-definition>
      <column-index>1</column-index>
      <auto-increment>false</auto-increment>
      <case-sensitive>false</case-sensitive>
      <currency>false</currency>
      <nullable>0</nullable>
      <signed>false</signed>
      <searchable>true</searchable>
      <column-display-size>45</column-display-size>
      <column-label>NOME</column-label>
      <column-name>NOME</column-name>
      <schema-name></schema-name>
      <column-precision>45</column-precision>
      <column-scale>0</column-scale>
      <table-name>UTENTI</table-name>
      <catalog-name>libro</catalog-name>
```

```
      <column-type>12</column-type>
      <column-type-name>VARCHAR</column-type-name>
    </column-definition>
    <column-definition>
      <column-index>2</column-index>
      <auto-increment>false</auto-increment>
      <case-sensitive>false</case-sensitive>
      <currency>false</currency>
      <nullable>0</nullable>
      <signed>false</signed>
      <searchable>true</searchable>
      <column-display-size>45</column-display-size>
      <column-label>COGNOME</column-label>
      <column-name>COGNOME</column-name>
      <schema-name></schema-name>
      <column-precision>45</column-precision>
      <column-scale>0</column-scale>
      <table-name>UTENTI</table-name>
      <catalog-name>libro</catalog-name>
      <column-type>12</column-type>
      <column-type-name>VARCHAR</column-type-name>
    </column-definition>
  </metadata>
  <data>
    <currentRow>
      <columnValue>Federico</columnValue>
      <columnValue>Paparoni</columnValue>
    </currentRow>
    <currentRow>
      <columnValue>John</columnValue>
      <columnValue>Doe</columnValue>
    </currentRow>
  </data>
```

```
</webRowSet>
```

Questo file XML è la rappresentazione del WebRowSet, dove vengono riportate tutte le informazioni in esso contenute. Il file XML può, in un classico scenario di utilizzo, essere archiviato su filesystem, magari per essere elaborato da altri processi, che dovranno semplicemente importarlo nel seguente modo:

```
WebRowSetImpl webRowSet = new WebRowSetImpl();
java.io.FileReader reader = new java.io.FileReader("test.xml");
webRowSet.readXml(reader);
```

EFFETTUARE TRANSAZIONI

È possibile realizzare transazioni in diversi modi. Uno dei modi che abbiamo già visto è quello di gestire la transazione attraverso i metodi *setAutoCommit()*, *commit()* e *rollback()*. Qui di seguito vediamo un esempio che utilizza questi metodi:

```
Class.forName("com.mysql.jdbc.Driver");
String url = "jdbc:mysql://localhost:3306/libro";
Connection connection = DriverManager.getConnection(url, "root",
                                                        "mysql");
connection.setAutoCommit(false);
...
...
//DOPO UNA SERIE DI OPERAZIONI EFFETTUIAMO UN SAVEPOINT
//AL QUALE POSSIAMO TORNARE SUCCESSIVAMENTE SE QUALCOSA NON
                                          VA PER IL VERSO GIUSTO
SavePoint savePoint=connection.setSavepoint();
try {
    ...
...
```

```
connection.commit();
}
catch(SQLException e) {
	connection.rollback(savePoint);
}
connection.setAutoCommit(true);
```

Come potete vedere, in questo caso, per gestire le nostre operazioni in maniera transazionale, abbiamo disabilitato l'auto commit. Poi abbiamo salvato ad un certo punto le operazioni fatte attraverso un *SavePoint*. Questo oggetto ci permette di memorizzare un certo numero di operazioni che sono state fatte sulla nostra connessione. In questo modo, se in seguito avviene qualcosa di errato, nel nostro codice, possiamo effettuare il rollback, tornando direttamente al punto che abbiamo salvato attraverso il metodo *rollback* di *Connection* che accetta come parametro un *SavePoint*. Nel caso in cui tutto proceda correttamente possiamo invece effettuare la commit. Alla fine di tutte le operazioni che abbiamo effettuato in questa "transazione" possiamo ripristinare l'auto commit. Diciamo che i *SavePoint* possono essere utilizzati come dei punti di backup della nostra transazione, che noi possiamo richiamare in base a quello che succede nel nostro programma. Oltre a questo metodo, Java mette a disposizione una API che ci permette di definire ad alto livello una transazione, *Java Transaction API* (JTA). Attraverso questa API possiamo gestire diverse risorse all'interno di una transazione. Una di queste risorse può essere tranquillamente un database oppure una coda JMS o una qualsiasi risorsa che permette di essere gestita attraverso JTA (di solito queste informazioni vengono riportate nella documentazione di un determinato prodotto). Per quanto riguarda un database dobbiamo prima di tutto avere a disposizione dei driver che permettono di ottenere un *XADataSource*, un'interfaccia che riesce a gestire la connessione al database come una risorsa all'interno di una transazione. Il classico esempio in cui possiamo utilizzare JTA è quello in cui, all'interno di un Application Server, la nostra applicazione deve ge-

stire diversi database in maniera transazionale. Possiamo immaginare di dover spostare delle informazioni da un database all'altro, magari dei soldi da un conto ad un altro. Chiaramente questa operazione deve essere eseguita all'interno di una transazione, perché se qualcosa va male non possiamo prelevare soldi da un conto senza inserirli da qualche altra parte. Vediamo, quindi, come possiamo realizzare questa funzionalità attraverso le JTA:

```
//INIZIAMO CON IL REPERIRE LE CONNESSIONI VERSO
//2 DIVERSI DATASOURCE, CONFIGURATI LATO SERVER PER ESSERE
                                          UTILIZZATI IN TRANSAZIONI

InitialContext ctx=new InitialContext();
DataSource
  banca1=(javax.sql.DataSource)ctx.lookup("java:comp/env/jdbc/Banca1");

DataSource
  banca2=(javax.sql.DataSource)ctx.lookup("java:comp/env/jdbc/Banca2");
Connection connessione1=banca1.getConnection();
Connection connessione2=banca2.getConnection();

//ORA DOBBIAMO OTTENERE TRAMITE JNDI LO USERTRANSACTION
//L'OGGETTO CHE CI PERMETTE DI INCAPSULARE LE OPERAZIONI CHE
                             FARANNO PARTE DELLA NOSTRA TRANSAZIONE

UserTransaction
   userTransation=(UserTransaction)ctx.lookup(URL_USER_TRANSACTION);
userTransaction.begin();
prendiSoldi(connessione1);
depositaSoldi(connessione2);
userTransaction.commit();
connessione1.close();
connessione2.close();
```

In questo esempio abbiamo immaginato di avere configurato sul nostro Application Server due diversi DataSource, che per la precisione sono *XADataSource*, ovvero che possono entrare a far parte di una transazione. Questa configurazione cambia per ogni Application Server, quindi per maggiori dettagli vi rimando alla guida di ognuno. Dopo che abbiamo ottenuto le due diverse connessioni, abbiamo recuperato tramite JNDI lo *UserTransaction*, che è l'oggetto utilizzabile dal punto di vista del client, il quale ci permette di inglobare diverse operazioni all'interno di una transazione. La vera e propria transazione è trasparente per il programmatore, non c'è cioè bisogno di realizzare tutta la comunicazione con le diverse entità in gioco. Di questo se ne occupa il TransactionManager, che è integrato nel nostro Application Server e che implementa il famoso Two Phase Commit (2PC) per garantire la transazione. Questo protocollo prevede che per ogni risorsa in gioco ci siano appunto due diverse fasi:

1. Nella prima fase viene inviato ad ogni risorsa la query da eseguire. Per andare avanti ogni risorsa deve rispondere con un *OK*
2. Nella seconda fase, il coordinatore chiede di effettuare la commit della query. Per dichiarare la transazione completata ogni risorsa deve inviare un *OK* al coordinatore

Come potete vedere dal precedente codice, non siamo dovuti intervenire sulla singola connessione per effettuare il commit, proprio perché la transazione viene gestita dal TransactionManager che è presente nel nostro ApplicationServer. In questo caso abbiamo inserito nella nostra transazione soltanto delle connessioni verso il database, ma potevamo inserire anche una coda JMS o un sistema ERP.

CONSIDERAZIONI

JDBC è un ottimo prodotto, che permette al programmatore Java di dialogare subito con un database. Anche le feature che offre dal punto di

vista delle API sono molte ed interessanti. In maniera dipendente dal progetto di cui ci stiamo occupando può essere utile o meno sviluppare la nostra applicazione utilizzando JDBC. Infatti, attualmente, esistono molti prodotti opensource (alcuni dei quali vedremo nei prossimi capitoli) che si basano su JDBC per offrire allo sviluppatore un'interfaccia ad alto livello e che sono una valida alternativa anche perché nella maggior parte dei casi velocizzano lo sviluppo.

HIBERNATE

Hibernate è una famosa libreria/framework Java, realizzata da Gavin King, che permette di avere un ORM (Object Relational Mapping) per il nostro software. Non stiamo parlando di un vero e proprio database ad oggetti ma di un tool che ci permette di avvicinare il paradigma ad oggetti di Java con quello relazionale dei database.

Figura 2.1: Object Relational Mapping

Un ORM come Hibernate cerca quindi di rendere trasparente allo sviluppatore la presenza di un database relazionale, cercando di effettuare un mapping Classe-Tabella. Hibernate è attualmente arrivato alla ver-

sione 3.2.5 ed è un progetto opensource molto vasto, che comprende al suo interno diversi “sottoprogetti” molto interessanti:

- **Hibernate Core**: Il cuore del progetto Hibernate
- **Hibernate Annotations**: Annotations che possono essere usate con Hibernate per evitare i file di mapping XML
- **Hibernate Entity Manager**: Implementazione delle Java Persistence API che si appoggia ad Hibernate Core. Questo progetto viene attualmente utilizzato di default in JBoss EJB 3.0
- **Hibernate Shards**: È un framework che ci permette di utilizzare Hibernate Core anche in quei casi in cui il nostro database viene partizionato, stravolto.
- **Hibernate Validator**: Serve per effettuare la validazione dei bean che stiamo per inserire/editare, utilizzando delle semplici annotations
- **Hibernate Search**: Framework che fornisce un’interessante API per effettuare ricerche nel nostro database, sfruttando le potenzialità di Apache Lucene (*http://lucene.apache.org*)
- **Hibernate Tools**: Plugin Ant/Eclipse per sviluppare le nostre applicazioni con Hibernate
- **NHibernate**: Porting .NET di Hibernate
- **JBoss Seam**: Framework web che utilizza Hibernate come “collante” tra JSF e EJB 3.0

Hibernate è compatibile con tutti i database che supportano JDBC, permettendo quindi di slegare il più possibile lo sviluppo della nostra applicazione dalla parte relativa al database. Questo già succedeva con JDBC, ma con un tool di ORM possiamo gestire la parte relativa ai dati della nostra applicazione in maniera ancora meno vincolante. Per effettuare delle ricerche Hibernate fornisce un potente linguaggio che è l’HQL (*Hibernate Query Language*), oltre a supportare l’SQL standard e le interessanti query realizzate con i *Criteria* che avremo modo di vedere successivamente.

PRIMI PASSI

Come avrete intuito, anche guardando la tantissima documentazione presente sull'homepage del progetto (*http://www.hibernate.org*), Hibernate è un progetto molto interessante e con molte cose da imparare.
Come per ogni libreria Java, anche in questo caso sarà necessario scaricare il JAR di questa libreria e includerlo nel nostro progetto.
Per iniziare ad utilizzare questo tool vedremo ora un esempio completo, andando poi in dettaglio su tutte le peculiarità di questo progetto nei successivi paragrafi. Il nostro punto di partenza sarà una semplice classe *Libri:*

```
package it.ioprogrammo.librodb.hibernate;

public class Libri implements java.io.Serializable {

    private Integer id;
    private String isbn;
    private String titolo;
    private String autore;
    private String categoria;

    public Libri() {
    }

    public Libri(String isbn, String titolo, String autore, String categoria) {
        this.isbn = isbn;
        this.titolo = titolo;
        this.autore = autore;
        this.categoria = categoria;
    }

    public Libri(String isbn, String titolo, String autore, String categoria,
            Set redazionis) {
```

```
        this.isbn = isbn;
        this.titolo = titolo;
        this.autore = autore;
        this.categoria = categoria;
        this.redazionis = redazionis;
    }

    public Integer getId() {
        return this.id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getIsbn() {
        return this.isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public String getTitolo() {
        return this.titolo;
    }

    public void setTitolo(String titolo) {
        this.titolo = titolo;
    }

    public String getAutore() {
        return this.autore;
```

```
	}

	public void setAutore(String autore) {
		this.autore = autore;
	}

	public String getCategoria() {
		return this.categoria;
	}

	public void setCategoria(String categoria) {
		this.categoria = categoria;
	}
}
```

La classe *Libri* è il solito JavaBean che possiamo utilizzare nella logica di business della nostra applicazione. Definiamo sul nostro database la tabella sulla quale verranno mappati gli oggetti di questa classe:

```
CREATE TABLE `libro`.`libri` (
 `ID` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
 `ISBN` VARCHAR(45) NOT NULL,
 `TITOLO` VARCHAR(45) NOT NULL,
 `AUTORE` VARCHAR(45) NOT NULL,
 `CATEGORIA` VARCHAR(45) NOT NULL,
 PRIMARY KEY (`ID`)
)
```

A questo punto, con le classiche connessioni JDBC avremmo creato una Connection verso il database e gestito l'inserimento/lettura/modifica/ cancellazione di questi dati attraverso Statement, ResultSet e tecnologie analoghe. Utilizzando Hibernate dobbiamo prima di tutto far capire a questo tool quale è il database con cui deve dialogare. Questo è possi-

bile attraverso il file di configurazione di Hibernate, che vediamo riportato qui di seguito

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
      <property
   name="hibernate.bytecode.use_reflection_optimizer">false</property>
      <property
 name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</prop
                                                                 erty>
      <property
               name="hibernate.connection.password">mysql</property>
      <property
 name="hibernate.connection.url">jdbc:mysql://127.0.0.1/libro</property
                                                                     >
      <property name="hibernate.connection.username">root</property>
      <property
 name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property
                                                                     >
      <property
    name="hibernate.current_session_context_class">thread</property>
      <mapping
            resource="it/ioprogrammo/librodb/hibernate/Libri.hbm.xml" />
   </session-factory>
</hibernate-configuration>
```

Questo file XML, denominato per default *hibernate.cfg.xml*, descrive al tool alcune cose che gli servono per accedere al database come url JDBC, driver JDBC, username e password. Viene inoltre spe-

cificato quale “dialetto” SQL utilizzare con la property *hibernate.dialect*. Esistono infatti molti “dialetti” SQL che cambiano sensibilmente la gestione di diverse cose all’interno del database.
Alla fine di questa configurazione abbiamo anche detto a Hibernate che esiste la definizione di una risorsa da mappare su questo database, utilizzando la seguente stringa

```
<mapping resource="it/ioprogrammo/librodb/hibernate/Libri.hbm.xml" />
```

Il file XML a cui ci siamo riferiti, *Libri.hbm.xml*, definisce il mapping su database per la nostra classe *Libri*. Attraverso questo file riusciamo quindi a spiegare ad Hibernate come deve far corrispondere le proprietà di un oggetto Libri alle colonne della tabella libri. Qui di seguito potete vedere la sua struttura:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
                                                    DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- Generated 29-gen-2008 15.21.25 by Hibernate Tools 3.2.0.CR1 -->
<hibernate-mapping>
  <class name="it.ioprogrammo.librodb.hibernate.Libri" table="libri"
                                                    catalog="libro">
    <id name="id" type="java.lang.Integer">
      <column name="ID" />
      <generator class="identity" />
    </id>
    <property name="isbn" type="string">
      <column name="ISBN" length="45" not-null="true" />
    </property>
    <property name="titolo" type="string">
      <column name="TITOLO" length="45" not-null="true" />
    </property>
```

```
        <property name="autore" type="string">
            <column name="AUTORE" length="45" not-null="true" />
        </property>
        <property name="categoria" type="string">
            <column name="CATEGORIA" length="45" not-null="true" />

        </property>
    </class>
</hibernate-mapping>
```

In *Libri.hbm.xml* possiamo vedere come viene riportato, proprietà per proprietà, il mapping da compiere. All'inizio del file c'è l'associazione tra nome della classe (comprensivo di package) e nome della tabella

```
<class name="it.ioprogrammo.librodb.hibernate.Libri" table="libri" cata
                                                            log="libro">
```

Per l'id della tabella *libri* viene associato un generatore di ID, che in questo caso è identità, che gestirà l'id della tabella *libri*. Successivamente vengono riportate tutte le proprietà con nome, tipologia e colonna da associare. A questo punto Hibernate sa come collegarsi al nostro database, come dialogarci e soprattutto come gestire il mapping tra la nostra classe ed una tabella. Per concludere questo paragrafo introduttivo vedremo, quindi, un semplice esempio dove andiamo a creare un oggetto della classe *Libri* e salvarlo sul database:

```
package it.ioprogrammo.librodb.hibernate;
import org.hibernate.Session;
import org.hibernate.SessionFactory;

import org.hibernate.cfg.Configuration;

```

```
public class PrimiPassi {
    public static void main(String[] args) {
        SessionFactory sessionFactory=new
                        Configuration().configure().buildSessionFactory();
        Session session=sessionFactory.getCurrentSession();

        session.beginTransaction();
        Libri libro=new Libri();
        libro.setAutore("Federico Paparoni");
        libro.setCategoria("IT/Database");
        libro.setIsbn("123457890");
        libro.setTitolo("Java e database");
        session.save(libro);
        session.getTransaction().commit();
    }

}
```

Nelle prime righe di questa classe abbiamo recuperato un oggetto *Session*, che permette di gestire una sessione con il nostro database. Di solito questo oggetto viene gestito a livello di Application Server o con altri Session Factory che ci permettono una gestione migliore quando stiamo sviluppando un progetto. Comunque, attraverso *Session*, riusciamo a far partire la transazione all'interno della quale inseriremo il nostro oggetto. Per fare ciò dobbiamo semplicemente costruire una nuova istanza di *Libri*, settare i vari parametri e poi richiamare il metodo *save()* di *Session*. Per concludere il nostro esempio dobbiamo chiamare la *commit()* e possiamo vedere che sul nostro database sarà presente una nuova riga nella tabella *libri*.
Hibernate definisce 3 diversi stati che il nostro oggetto può avere:

- **Transient**: L'oggetto che vogliamo utilizzare è stato inizializzato ma non è ancora collegato in alcun modo con una sessione Hiber-

Figura 2.2: Nuova entry nel nostro database

nate, quindi non esiste un suo mapping sul database e i possibili cambiamenti che facciamo su questo oggetto non si ripercuotono sul database

- **Persistent**: L'oggetto è stato associato ad una sessione Hibernate, salvandolo su database attraverso il metodo *save()* oppure effettuando una query. In questo stato l'oggetto ha una corrispondenza nel nostro database
- **Detached**: L'oggetto, che abbiamo precedentemente ricavato da una sessione Hibernate, ora è slegato da essa quindi qualsiasi cosa succede a questo oggetto non avrà alcuna conseguenza sul database, a meno che l'oggetto non venga ricollegato ad una nuova sessione

Gli oggetti che si trovano nello stato *Transient* possono passare allo stato *Persistent* utilizzando i seguenti metodi della classe

org.hibernate.Session: *save()*, *persist()* e *saveOrUpdate()*. In questo modo viene effettuato il mapping su database. Se invece abbiamo un oggetto che è stato recuperato dal database, questo è già nello stato *Persistent*, mentre se chiudiamo la sessione che lo ha recuperato allora entriamo nello stato *Detached*. Nei prossimi paragrafi vedremo i diversi modi che Hibernate offre per effettuare la ricerca sul nostro database. In seguito approfondiremo le questioni relative a mapping più complessi di quello che abbiamo ora realizzato.

HQL E SQL

Uno dei metodi che Hibernate offre per effettuare delle ricerche è quello di utilizzare l'HQL, *Hibernate Query Language*. Questo linguaggio permette di definire in maniera simile all'SQL, ma Object Oriented, una query da effettuare. Vediamo un esempio dove utilizziamo questo linguaggio per recuperare tutte le istanze di *Libri* dal database:

```
package it.ioprogrammo.librodb.hibernate;

import java.util.Iterator;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class QueryHQL {

    public static void main(String[] args) {
        SessionFactory sessionFactory=new
                    Configuration().configure().buildSessionFactory();
        Session session=sessionFactory.openSession();
        String queryHQL="from Libri";
```

```
        Query query=session.createQuery(queryHQL);
        Iterator iterator=query.iterate();
        Libri temp;
        while(iterator.hasNext()) {
            temp=(Libri)iterator.next();
            System.out.println("Titolo: "+temp.getTitolo());
            System.out.println("Autore: "+temp.getAutore());
            System.out.println("ISBN: "+temp.getIsbn());
        }
        session.close();
    }

}
```

La query HQL che ci serve per effettuare quello che in SQL sarebbe un *"SELECT * FROM LIBRI"* è *"from Libri"*. In questo caso però, *Libri* non è la tabella ma la classe a cui siamo interessati. Praticamente con l'HQL noi abbiamo un linguaggio per effettuare le query sul nostro database, ma possiamo continuare a ragionare nell'ottica OO della nostra applicazione. Questo linguaggio conserva tutte le caratteristiche standard dell'SQL, come ad esempio le clausole "*WHERE*" e "*ORDER BY*". Ad esempio se fossimo interessati a recuperare tutti i libri che hanno come categoria "*IT/Software Engineering*" e ordinarli in base al titolo, la nostra query HQL sarebbe la seguente:

```
from Libri libri where libri.categoria='IT/Software Engineering' order by
                                                              libri.titolo
```

L'esecuzione di queste query viene demandata a Hibernate attraverso il metodo *createQuery()* di *Session*, che prende come argomento una stringa contenente la query. Come risultato abbiamo un oggetto *Query*, che possiamo utilizzare attraverso un classico Iterator. E' possibile anche gestire il join di diversi oggetti tramite HQL, collegamenti che possono

essere definiti sempre tramite Hibernate e che vedremo in un paragrafo successivo. Hibernate, nonostante sia un framework per l'Object Relational Mapping e gestisca nativamente tutte le sessioni di dialogo con il nostro database, ci permette di utilizzare l'SQL. A primo impatto si potrebbe pensare che questo sia un controsenso, ma talvolta invece può essere necessario utilizzare l'SQL nativo del nostro database, per utilizzare magari funzioni specifiche che non potremmo utilizzare in altre maniere. Per utilizzare l'SQL standard dobbiamo utilizzare il metodo *createSQLQuery()* di *Session*, passando come argomento la classica query SQL:

```
package it.ioprogrammo.librodb.hibernate;

import java.util.Iterator;
import java.util.List;

import org.hibernate.Hibernate;
import org.hibernate.SQLQuery;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class QuerySQLNativo {

    public static void main(String[] args) {
        SessionFactory sessionFactory=new
                        Configuration().configure().buildSessionFactory();
        Session session=sessionFactory.openSession();
        String querySQL="SELECT * FROM LIBRI";
        SQLQuery query=session.createSQLQuery(querySQL);
        query=query.addScalar("TITOLO",Hibernate.STRING);
query=query.addScalar("AUTORE",Hibernate.STRING);
        query=query.addScalar("ISBN",Hibernate.STRING);
```

```
                List list=query.list();
                Iterator iterator=list.iterator();
                Object[] temp;
                while(iterator.hasNext()) {
                        temp=(Object[])iterator.next();
                        for(int i=0;i<temp.length;i++) {
                                System.out.println(temp[i]);
                        }
                }
                session.close();
        }

}
```

Con la query SQL che abbiamo definito, siamo in grado di recuperare tutti valori della tabella *libri*. Utilizzando il metodo *addScalar()* di *Query* abbiamo definito quali campi sono di nostro interesse. Quello che ci viene restituito alla fine è una lista che al suo interno ha array di Object. Ogni array rappresenta una riga con le varie colonne che abbiamo selezionato. Chiaramente quando utilizziamo altre tipologie di ricerca con Hibernate siamo facilitati dalla gestione Object Oriented dei dati. Diciamo che questa tipologia di ricerca è stata inclusa nel progetto per quei casi particolari in cui è necessario utilizzare l'SQL nativo del database.

LE API CRITERIA

I Criteria sono una delle API più interessanti che Hibernate mette a disposizione, perché permette di definire una query, modellandola da un punto di vista Object Oriented.
Proprio per questo esiste un package dedicato, *org.hibernate.criterion*, che include al suo interno tutta una serie di interfacce e classi che servono per modellare le nostre query.
Vediamo, quindi, un esempio in cui utilizziamo i *Criteria* di Hibernate:

```
package it.ioprogrammo.librodb.hibernate;

import java.util.Iterator;
import java.util.List;

import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.criterion.Order;

public class QueryCriteria {

    public static void main(String[] args) {
        Criteria criteria;
        Libri temp;
        SessionFactory sessionFactory=new
                    Configuration().configure().buildSessionFactory();
        Session session=sessionFactory.openSession();

        criteria = session.createCriteria(Libri.class);
        criteria.setMaxResults(20);
        criteria.addOrder(Order.asc("titolo"));
        List lista = criteria.list();

        Iterator iterator=lista.iterator();
        while(iterator.hasNext()) {
            temp=(Libri)iterator.next();
            System.out.println(temp.getTitolo()+" -
                                        "+temp.getCategoria());
        }
        session.close();
    }
```

```
}
```

Per effettuare una semplice ricerca con i *Criteria* abbiamo utilizzato il metodo *createCriteria()* di *Session*, passando come parametro la classe a cui siamo interessati. In questo modo è come se avessimo fatto una "*SELECT * FROM LIBRI*", perché non abbiamo specificato alcun vincolo. L'unica cosa particolare che abbiamo fatto è decidere il numero di risultati da recuperare utilizzando il metodo *setMaxResults()*. Inoltre abbiamo deciso che vogliamo i risultati della nostra query, ordinati in base alla proprietà "titolo" grazie al metodo *addOrder()* di *Criteria*.
Già da questo primo esempio potete vedere come sia elegante e semplice il metodo di ricerca basato sui Criteria. Chiaramente, le possibilità offerte da questa API vanno oltre la semplice select. Molto spesso succede di dover modellare nel nostro codice una query in base a determinati parametri che vengono richiesti. Per avvicinarci al nostro esempio, potremmo aver bisogno di elencare tutti i libri che fanno parte di una certa categoria e che contengono al loro interno una stringa. Per fare ciò non dobbiamo far altro che aggiungere al Criteria che stiamo utilizzando alcune restrizioni, come potete vedere nel seguente codice:

```
Criterion crit1=Restrictions.eq("categoria", "IT/Database");
Criterion crit2=Restrictions.eq("categoria", "IT/Software Development");
Criterion crit3=Restrictions.or(crit1, crit2);
criteria.add(crit3);
List lista = criteria.list();
```

Attraverso i metodi statici della classe *org.hibernate.criterion.Restrictions* abbiamo definito 3 diversi *Criterion*, ovvero 3 diverse restrizioni da applicare alla nostra ricerca tramite Criteria. Nel primo abbiamo richiesto che il parametro categoria fosse uguale a "*IT/Database*", nel secondo a "*IT/Software Development*". Con il terzo *Criterion* abbiamo invece accorpato i due precedenti utilizzando un OR logico. In questo modo la

nostra ricerca ritornerà risultati che soddisfano la prima o la seconda condizione. Questo è solo una delle tante cose che ci permette di modellare la query attraverso i Criteria. Immaginiamo, ad esempio, di voler realizzare una serie di ricerche che devono essere fatte per ottenere i libri che iniziano con una particolare lettera. Questo è possibile con il modello *like()* di *Restrictions*, al quale passiamo come parametri quale variabile, cosa ricercare e in quale modalità:

```
Criterion letterA=Restrictions.like("titolo","A",MatchMode.START);
Criterion letterB=Restrictions.like("titolo","B",MatchMode.START);
Criterion letterC=Restrictions.like("titolo","C",MatchMode.START);
```

Vedendo in dettaglio tutti i vari metodi di Restrictions si intuisce che, utilizzando i Criteria, possiamo creare un semplice ma potente sistema per effettuare query sul nostro database.

QUERY BY EXAMPLE

Passiamo ora ad un ulteriore metodo che ci permette di effettuare delle query utilizzando Hibernate. Con i Criteria possiamo definire in maniera programmatica, e non attraverso delle stringhe, una ricerca da effettuare. Basandoci su questa API possiamo anche effettuare una ricerca passando come parametro un vero e proprio oggetto, della stessa tipologia che ci aspettiamo come risultato, valorizzando le proprietà che verranno poi utilizzate per effettuare la ricerca. Praticamente, se vogliamo cercare un libro con un determinato titolo, possiamo anche creare un nuovo oggetto, settare solo il titolo e poi avviare la ricerca.
Hibernate lo analizzerà e ricercherà soltanto i mapping che possono corrispondere ad un oggetto con quel titolo:

```
package it.ioprogrammo.librodb.hibernate;
import java.util.Iterator;
import java.util.List;
```

```
import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.criterion.Example;

public class QueryByExample {

    public static void main(String[] args) {
        Criteria criteria;
        Libri temp;
        SessionFactory sessionFactory=new
                    Configuration().configure().buildSessionFactory();
        Session session=sessionFactory.openSession();
        Libri libro=new Libri();
        libro.setTitolo("Java e database");

        Example libroExample;
        libroExample=Example.create(libro);
        libroExample.ignoreCase();

        criteria = session.createCriteria(Libri.class);
        criteria.add(libroExample);
        List lista=criteria.list();
        Iterator iterator=lista.iterator();
        while(iterator.hasNext()) {
            temp=(Libri)iterator.next();
            System.out.println(temp.getTitolo()+" -
                                        "+temp.getCategoria());
        }
        session.close();
    }

```

```
}
```

Tale metodologia viene definita "*query by example*", poichè andiamo a realizzare un oggetto *Example*, passandolo come parametro d'esempio da ricercare. In seguito viene creato un *Criteria* che si basa sulla classe *Libri* e andiamo ad aggiungere questo oggetto d'esempio. In questo modo abbiamo realizzato la nostra ricerca utilizzando un oggetto come modello.

ASSOCIAZIONI

Incominciamo ora a considerare un nuovo database, dove abbiamo diverse entità che entrano in gioco. Fino al precedente paragrafo abbiamo visto solo come mappare una singola classe con una tabella. Ora vediamo come viene modificato il nostro database, introducendo nuove tabelle con delle foreign key. Per il momento avevamo solo la tabella *libri*, ad un certo punto decidiamo di cambiare il nostro database, gestendo anche l'informazione relativa all'autore. Definiamo quindi due diverse tabelle, *autore* e *libri*, che gestiranno meglio questa informazione. Praticamente l'informazione che era presente prima nella tabella *libri*, sotto la colonna *AUTORE*, adesso diventa importante per il dominio della nostra applicazione e quindi dobbiamo gestirla in maniera differente.

Ecco quindi lo script SQL per generare il database che dobbiamo utilizzare:

```
CREATE TABLE `libro`.`autore` (
 `ID` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
 `NOME` VARCHAR(45) NOT NULL,
 `COGNOME` VARCHAR(45) NOT NULL,
 `EMAIL` VARCHAR(45) NOT NULL,
 PRIMARY KEY (`ID`)
)
ENGINE = InnoDB;
```

```
CREATE TABLE `libro`.`libri` (
 `ID` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
 `ISBN` VARCHAR(45) NOT NULL,
 `TITOLO` VARCHAR(45) NOT NULL,
 `CATEGORIA` VARCHAR(45) NOT NULL,
 `ID_AUTORE` INTEGER UNSIGNED NOT NULL,
 PRIMARY KEY (`ID`),
 CONSTRAINT `FK_libri_1` FOREIGN KEY `FK_libri_1` (`ID_AUTORE`)
   REFERENCES `autore` (`ID`)
   ON DELETE CASCADE
   ON UPDATE CASCADE
)
ENGINE = InnoDB;
```

La seconda cosa che dobbiamo poi cambiare, è il file di mapping della classe *Libri*. Ora non avremo più una semplice proprietà relativa ad una colonna, ma una relazione di tipo molti ad uno, ovvero tanti libri possono essere associati a un autore:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
                                                    DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="it.ioprogrammo.librodb.hibernate.Libri" table="libri"
                                                    catalog="libro">
        <id name="id" type="java.lang.Integer">
            <column name="ID" />
            <generator class="identity" />
        </id>
        <many-to-one name="autore"
            class="it.ioprogrammo.librodb.hibernate.Autore" fetch="select">
            <column name="ID_AUTORE" not-null="true" />
```

```
        </many-to-one>
        <property name="isbn" type="string">
            <column name="ISBN" length="45" not-null="true" />
        </property>
        <property name="titolo" type="string">
            <column name="TITOLO" length="45" not-null="true" />
        </property>
        <property name="categoria" type="string">
            <column name="CATEGORIA" length="45" not-null="true" />
        </property>
    </class>
</hibernate-mapping>
```

Il tag *<many-to-one>* ci permette, appunto, di definire questo nuovo tipo di mapping, indicando a quale classe questa associazione si riferisce e quale colonna del database è interessata. Nella classe relativa non avremo più un oggetto di tipo *String*, ma il nuovo oggetto *Autore:*

```
private Autore autore;
```

Vediamo ora questo nuovo mapping. In questo caso vedremo un nuovo costrutto di Hibernate, l'insieme (*set*). Questo serve per gestire la relazione che esiste tra queste due tabelle, in maniera diametralmente opposta a quella relativa ai libri.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
                                                    DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="it.ioprogrammo.librodb.hibernate.Autore"
                                table="autore" catalog="libro">
        <id name="id" type="java.lang.Integer">
```

```
            <column name="ID" />
            <generator class="identity" />
        </id>
        <property name="nome" type="string">
            <column name="NOME" length="45" not-null="true" />
        </property>
        <property name="cognome" type="string">
            <column name="COGNOME" length="45" not-null="true" />
        </property>
        <property name="email" type="string">
            <column name="EMAIL" length="45" not-null="true" />
        </property>
        <set name="libris" inverse="true">
            <key>
                <column name="ID_AUTORE" not-null="true" />
            </key>
            <one-to-many class="it.ioprogrammo.librodb.hibernate.Libri" />
        </set>
    </class>
</hibernate-mapping>
```

L'autore può chiaramente aver scritto più libri, visto che la tabella *AUTORI* ha una foreign key su libri. Questa situazione viene gestita con un insieme, dove viene specificato che la relazione è uno a molti, attraverso il tag *<one-to-many>*. Questo nuovo mapping deve essere aggiunto al nostro file *hibernate.cfg.xml:*

```
<mapping resource="it/ioprogrammo/librodb/hibernate/Autore.hbm.xml"
                                                                  />
```

Per completare tutto il nostro viaggio nelle associazioni adesso dobbiamo vedere anche la classe *Autore*, dove avremo un *Set* (*java.util.HashSet* per la precisione) che gestisce l'insieme di libri associati ad un determi-

nato autore.

```
package it.ioprogrammo.librodb.hibernate;

import java.util.HashSet;
import java.util.Set;

public class Autore implements java.io.Serializable {

    private Integer id;
    private String nome;
    private String cognome;
    private String email;
    private Set libris = new HashSet(0);

    public Autore() {
    }

    public Autore(String nome, String cognome, String email) {
        this.nome = nome;
        this.cognome = cognome;
        this.email = email;
    }

    public Autore(String nome, String cognome, String email, Set libris) {
        this.nome = nome;
        this.cognome = cognome;
        this.email = email;
        this.libris = libris;
    }
    public Integer getId() {
        return this.id;
    }
```

```
    public void setId(Integer id) {
        this.id = id;
    }

    public String getNome() {
        return this.nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getCognome() {
        return this.cognome;
    }

    public void setCognome(String cognome) {
        this.cognome = cognome;
    }

    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Set getLibris() {
        return this.libris;
    }

    public void setLibris(Set libris) {
```

```
            this.libris = libris;
      }

}
```

Il funzionamento di questo nuovo mapping, a livello di creazione/ricerca/salvataggio/cancellazione è simile al precedente. Chiaramente, però, non possiamo ora creare un libro ex-novo senza collegare ad esso un oggetto *Autore*. Nel codice seguente vediamo come creare un oggetto *Autore*, associarlo a un autore già esistente e salvare il tutto:

```
package it.ioprogrammo.librodb.hibernate;

import java.util.Iterator;
import java.util.List;

import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
import org.hibernate.criterion.Example;

public class TestAssociazioni {

      public static void main(String[] args) {
            SessionFactory sessionFactory=new
                              Configuration().configure().buildSessionFactory();
            Session session=sessionFactory.getCurrentSession();
            session.beginTransaction();
            Autore autore=new Autore();
            autore.setNome("Federico");
            autore.setCognome("Paparoni");
            Example autoreExample;
```

```
            autoreExample=Example.create(autore);
            Criteria criteria = session.createCriteria(Autore.class);
            criteria.add(autoreExample);
            autore=(Autore)criteria.uniqueResult();
            Libri libro1=new Libri();
            libro1.setAutore(autore);
            libro1.setCategoria("IT/Database");
            libro1.setIsbn("1234567890");
            libro1.setTitolo("Java e database");
            Libri libro2=new Libri();
            libro2.setAutore(autore);
            libro2.setCategoria("IT/OS");
            libro2.setIsbn("0987654321");
            libro2.setTitolo("Operating Systems");
            session.save(libro1);
            session.save(libro2);
            session.getTransaction().commit();
      }
}
```

In una prima fase abbiamo cercato l'autore, già presente sul database, utilizzando una ricerca By Example. Successivamente abbiamo creato due nuovi libri e, dopo aver settato l'autore trovato, li abbiamo memorizzati su database. Il risultato di questo programma lo potete vedere nella seguente immagine, dove sono presenti due entry nella tabella libri, con *ID_AUTORE* uguale.
Questo che abbiamo visto non è l'unico modo in cui Hibernate gestisce le associazioni/collezioni, diciamo che è quello più vicino al mondo relazionale. Chiaramente, anche dal punto di vista delle ricerche, possiamo sfruttare le associazioni presenti tra diversi oggetti.
Ad esempio, se volessimo recuperare dal database tutti gli autori che hanno scritto libri di una particolare categoria potremmo definire il seguente *Criteria:*

MySQL Query Browser - Connection: root@127.0.0.1:3306 / libro

File Edit View Query Script Tools Window MySQL Enterprise Help

Go back Next Refresh

```
SELECT * FROM libri l;
```

Resultset 1

ID	ISBN	TITOLO	CATEGORIA	ID_AUTORE
1	1234567890	Java e database	IT/Database	1
2	0987654321	Operating Systems	IT/OS	1

Figura 2.3: Il risultato dell’associazione che abbiamo realizzato

```
session.createCriteria(Autore.class)
.createCriteria("libris", "libri")
.add(Expression.eq("libri.categoria", "IT/Database"))
.list();
```

Utilizzando il metodo *createCriteria(),* in questo caso, abbiamo incominciato ad inserire delle restrizioni sull’associazione che *Autore* ha con *Libri*. A livello pratico è come fare una join tra le due tabelle, specificando un certo valore che ci permette di ottenere i risultati voluti. Quando affrontiamo questo argomento dobbiamo fare molta attenzione ai diversi modi in cui poter caricare i dati relativi alle associazioni, perché giustamente potremmo avere dei problemi di performance. Hibernate per risolvere questo problema offre il meccanismo del lazy fetching, che ci permette di caricare in un secondo momento i dati di cui abbiamo bisogno. Per questa tematica e per altre che possono aiutare il tuning della vostra applicazione con Hibernate vi rimando alla documentazione ufficiale.

EREDITARIETA’

Hibernate è un framework che ci permette di mappare il nostro mondo Object Oriented in un sistema relazionale come quello dei database. Oltre alle associazioni, una delle cose che possono essere più di diffici-

li da trasformare in un modello relazionale è l'ereditarietà, la gerarchia che abbiamo tra le diverse classi che vengono utilizzate nel nostro progetto. Essendo un ORM, Hibernate ha chiaramente delle soluzioni anche per questo problema, infatti ci sono diversi modi in cui possiamo risolvere questo problema. Ora vedremo 3 diversi metodi, che vengono elencati qui di seguito

- Una tabella per sottoclasse
- Una tabella per gerarchia di classe
- Una tabella per classe concreta

Vediamo, quindi, la prima situazione, in cui abbiamo una classe principale, dalla quale vengono estese 3 diverse classi che utilizzeremo nel nostro progetto. Nella prossima immagine potete vedere le classi d'esempio che abbiamo utilizzato:

Figura 2.3: Primo esempio di ereditarietà

In questo esempio abbiamo una classe *base*, *veicolo*

```
package it.ioprogrammo.librodb.hibernate;

public class Veicolo {
	int id;
```

```
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```

e tre sottoclassi: *Automobile*, *Camion* e *Moto*

```
package it.ioprogrammo.librodb.hibernate;

public class Automobile extends Veicolo {
    String marca;

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }
}
...
...
package it.ioprogrammo.librodb.hibernate;
public class Camion extends Veicolo {
    int portata;

    public int getPortata() {
        return portata;
```

```
    }

    public void setPortata(int portata) {
        this.portata = portata;
    }
}
...
...
package it.ioprogrammo.librodb.hibernate;

public class Moto extends Veicolo {
    int cilindrata;

    public int getCilindrata() {
        return cilindrata;
    }

    public void setCilindrata(int cilindrata) {
        this.cilindrata = cilindrata;
    }
}
```

In questo tipo di situazione vogliamo creare un mapping che genera una tabella per ogni sottoclasse, anche perché non ci sono valori da mappare nella classe padre *Veicolo* oltre all'id. Quest'ultimo verrà mappato in una tabella a cui tutte le tabelle relative alle sottoclassi dovranno far riferimento. Per fare questo dobbiamo creare una tabella *VEICOLO* che avrà il solo *ID*, mentre le altre tabelle avranno tutti i valori di loro competenza, con una colonna *ID* che sarà un Foreign Key legata alla tabella *ID* di *VEICOLO*. Riportiamo ora l'SQL che serve a creare queste quattro tabelle sul nostro database MySQL:

```
CREATE TABLE `libro`.`veicolo` (
```

```
 `ID` INTEGER UNSIGNED NOT NULL DEFAULT NULL AUTO_INCREMENT,
 PRIMARY KEY (`ID`)
)
ENGINE = InnoDB;

CREATE TABLE `libro`.`automobile` (
 `ID` INTEGER UNSIGNED NOT NULL,
 `MARCA` VARCHAR(45) NOT NULL,
 CONSTRAINT `FK_automobile_1` FOREIGN KEY `FK_automobile_1` (`ID`)
  REFERENCES `veicolo` (`ID`)
  ON DELETE CASCADE
  ON UPDATE CASCADE
)
ENGINE = InnoDB;

CREATE TABLE `libro`.`camion` (
 `ID` INTEGER UNSIGNED NOT NULL,
 `PORTATA` INTEGER UNSIGNED NOT NULL,
 CONSTRAINT `FK_camion_1` FOREIGN KEY `FK_camion_1` (`ID`)
  REFERENCES `veicolo` (`ID`)
  ON DELETE CASCADE
  ON UPDATE CASCADE
)
ENGINE = InnoDB;

CREATE TABLE `libro`.`moto` (
 `ID` INTEGER UNSIGNED NOT NULL,
 `CILINDRATA` INTEGER UNSIGNED NOT NULL,
 CONSTRAINT `FK_moto_1` FOREIGN KEY `FK_moto_1` (`ID`)
  REFERENCES `veicolo` (`ID`)
  ON DELETE CASCADE
  ON UPDATE CASCADE
)
```

```
ENGINE = InnoDB;
```

Ora che abbiamo definito nei due diversi mondi, relazionale e OO, come gestire le nostre informazioni, dobbiamo istruire Hibernate su come gestire il mapping. Per fare ciò dovremo come sempre definire un file XML di mapping:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
                                                        DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class name="it.ioprogrammo.librodb.hibernate.Veicolo"
                          table="veicolo" polymorphism="implicit">
    <id name="id" column="ID">
      <generator class="native"/>
    </id>
    <joined-subclass name="it.ioprogrammo.librodb.hibernate.Automobile"
                                                table="automobile">
      <key column="ID"/>
      <property name="marca" column="marca" type="java.lang.String"
                                                                  />
    </joined-subclass>
    <joined-subclass name="it.ioprogrammo.librodb.hibernate.Camion"
                                                    table="camion">
      <key column="ID"/>
      <property name="portata" column="portata" type="int" />
    </joined-subclass>
    <joined-subclass name="it.ioprogrammo.librodb.hibernate.Moto"
                                                      table="moto">
      <key column="ID"/>
      <property name="cilindrata" column="cilindrata" type="int" />
```

```
   </joined-subclass>
  </class>
</hibernate-mapping>
```

Questo file di mapping introduce delle novità rispetto a quelli che abbiamo visto fino a questo punto. All'interno della classe *Veicolo*, mappata sulla rispettiva tabella, possiamo vedere che ci sono una serie di sottoclassi definite con il tag *joined-subclass*. In questo modo Hibernate è a conoscenza dei collegamenti tra classi e tabelle, altresì è conscio che quando si inserisce un nuovo *Veicolo* nella tabella omonima, successivamente dovrà fare una seconda insert nella tabella relativa alla sottoclasse, utilizzando l'ID della tabella *veicolo*. Vediamo, quindi, un esempio di codice da utilizzare con questo mapping:

```
package it.ioprogrammo.librodb.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class AssociazioneUno {

    public static void main(String[] args) {
        SessionFactory sessionFactory=new
                        Configuration().configure().buildSessionFactory();
        Session session=sessionFactory.getCurrentSession();
        session.beginTransaction();

        Moto moto=new Moto();
        moto.setCilindrata(250);
        Camion camion=new Camion();
        camion.setPortata(200);
        Automobile auto=new Automobile();
```

```
            auto.setMarca("ROVER");

            session.save(moto);
            session.save(camion);
            session.save(auto);
            session.getTransaction().commit();
      }

}
```

La classe d'esempio qui riportata inserisce tre diversi veicoli nel nostro database. Dietro le quinte Hibernate esegue i seguenti comandi SQL (che possiamo vedere dal log settando a *true* la proprietà *hibernate.show_sql* nel file di configurazione)

```
Hibernate: insert into veicolo values ( )
Hibernate: insert into moto (cilindrata, ID) values (?, ?)
Hibernate: insert into veicolo values ( )
Hibernate: insert into camion (portata, ID) values (?, ?)
Hibernate: insert into veicolo values ( )
Hibernate: insert into automobile (marca, ID) values (?, ?)
```

Come avevamo già preannunciato, Hibernate inserisce prima una nuova riga nella tabella veicolo e successivamente effettua l'insert nella tabella relativa alla sottoclasse, riportando l'ID della nuova riga di veicolo appena creata. Passiamo ora a una diversa soluzione per questo problema, quella che ci suggerisce di utilizzare una tabella per ogni gerarchia di classe. Praticamente, utilizzando questo metodo, dovremmo avere una tabella che mappa concettualmente tutta una nostra gerarchia. Da un certo punto di vista questa potrebbe essere la soluzione più pratica, anche se come primo problema abbiamo che all'interno di questa tabella non tutte le colonne saranno valorizzate, visto che saranno presenti tante colonne quanti sono i vari parametri di tutte le sottoclassi del-

la gerarchia.

Figura 2.5: Secondo esempio di ereditarietà

Per capire meglio anche questa soluzione, partiamo da un altro esempio, con una classe padre *Telefono:*

```
package it.ioprogrammo.librodb.hibernate;

public class Telefono {
    int id;
    String numero;
    String discriminator;

    public int getId() {
        return id;
    }
    public void setId(int id) {
```

```
            this.id = id;
      }
      public String getNumero() {
            return numero;
      }
      public void setNumero(String numero) {
            this.numero = numero;
      }
      public String getDiscriminator() {
            return discriminator;
      }
      public void setDiscriminator(String discriminator) {
            this.discriminator = discriminator;
      }
}
```

e due classi che la estendono, *TelefonoFisso* e *TelefonoCellulare:*

```
package it.ioprogrammo.librodb.hibernate;

public class TelefonoFisso extends Telefono {
      String distretto;

      public String getDistretto() {
            return distretto;
      }

      public void setDistretto(String distretto) {
            this.distretto = distretto;
      }
}
...
...
```

```
package it.ioprogrammo.librodb.hibernate;

public class TelefonoCellulare extends Telefono {
      String pianoTariffario;

      public String getPianoTariffario() {
            return pianoTariffario;
      }

      public void setPianoTariffario(String pianoTariffario) {
            this.pianoTariffario = pianoTariffario;
      }
}
```

Ognuna delle due classi figlie ha un parametro particolare che la caratterizza e che l'altra classe non deve gestire. Come abbiamo detto, in questa metodologia vogliamo utilizzare una sola tabella per mappare tutta una gerarchia di classi. Per definire la tabella sul nostro database, dobbiamo renderci conto che ci saranno tutti i valori possibili relativi alla gerarchia e in più una colonna che servirà a Hibernate per effettuare la distinzione sulla diversa tipologia di classe. Vediamo la definizione di questa tabella:

```
CREATE TABLE `libro`.`telefono` (
 `ID` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
 `DISCRIMINATOR` VARCHAR(45) NOT NULL,
 `NUMERO` VARCHAR(45) NOT NULL,
 `PIANOTARIFFARIO` VARCHAR(45),
 `DISTRETTO` VARCHAR(45),
 PRIMARY KEY (`ID`)

)
ENGINE = InnoDB;
```

Abbiamo inserito la colonna *DISCRIMINATOR* per capire a livello di entry quale sia la classe di pertinenza. Chiaramente non lo dovremo fare noi a mano, ma sarà Hibernate che in base al file di mapping che definiremo, riempirà questa colonna con il valore relativo alla rispettiva classe.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
                                                          DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
 <class name="it.ioprogrammo.librodb.hibernate.Telefono"
                    table="TELEFONO" polymorphism="implicit">
   <id name="id" column="ID">
     <generator class="native"/>
   </id>
   <discriminator>
     <column name="DISCRIMINATOR"/>
   </discriminator>
   <property name="numero" column="NUMERO"
                                   type="java.lang.String" />

   <subclass name="it.ioprogrammo.librodb.hibernate.TelefonoCellulare"
                      discriminator-value="TelefonoCellulare">
         <property name="pianoTariffario" column="PIANOTARIFFARIO"
                                   type="java.lang.String" />
   </subclass>

   <subclass name="it.ioprogrammo.librodb.hibernate.TelefonoFisso"
                         discriminator-value="TelefonoFisso">
         <property name="distretto" column="DISTRETTO"
                                   type="java.lang.String" />
```

```
    </subclass>
  </class>
</hibernate-mapping>
```

In questo mapping abbiamo definito il tag *<discriminator>*, settandogli il valore della colonna *DISCRIMINATOR*. Così facendo, Hibernate per capire con quale tipo di classe sta operando dovrà far riferimento al valore di quella colonna. Nel momento in cui andiamo a salvare un nuovo oggetto, Hibernate utilizzerà il valore di discriminator che abbiamo definito per la classe di quell'oggetto. Continuando a vedere il file di mapping possiamo notare che in questo caso le classi, tranne quella padre, vengono definite all'interno di tag *<subclass>*, dove vengono riportate le proprietà delle singole classi. Oltre a questo viene inserito come attributo di *subclass* il *discriminator-value*, che sarà il valore utilizzato per discriminare tra i diversi tipi di sottoclasse definiti. Vediamo quindi un esempio che inserirà due diversi oggetti, settando i relativi attributi di classe, lasciando vuoti nella riga quelli che non sono di sua pertinenza e settando il valore del discriminator:

```
SessionFactory sessionFactory=new
                        Configuration().configure().buildSessionFactory();
Session session=sessionFactory.getCurrentSession();
session.beginTransaction();

TelefonoCellulare cell=new TelefonoCellulare();
cell.setNumero("12345678909");
cell.setPianoTariffario("Piano famiglia");
TelefonoFisso fisso=new TelefonoFisso();
fisso.setNumero("123123123123");
fisso.setDistretto("Roma");

session.save(cell);
session.save(fisso);
```

```
session.getTransaction().commit();
```

Anche in questo caso, se abbiamo configurato tutto per bene, Hibernate lavorerà dietro le quinte per organizzare gli inserimenti nella singola tabella *TELEFONO*, settando gli opportuni valori. La terza via che possiamo intraprendere per mappare una gerarchia di classi, ovvero quella in cui andiamo a utilizzare una tabella per ogni classe concreta del nostro dominio. In questo caso quando andiamo a parlare di classe "concreta", parlando di gerarchia di classi, ci riferiamo alle classi figlie non astratte che utilizzeremo nel nostro progetto. Infatti con questa ultima tecnica avremo una tabella per ognuna di queste classi. La classe "non concreta" potrebbe essere, ad esempio, una classe padre che abbiamo definito astratta, come la seguente:

```
package it.ioprogrammo.librodb.hibernate;

public abstract class Programmi {
	int id;
	String nome;
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}
}
```

Nel codice, ai fini della logica del nostro programma, abbiamo deciso di definire questa classe astratta perché avremo bisogno di utilizzare solo le classi figlie *Videogiochi* e *OfficeSoftware*

```
package it.ioprogrammo.librodb.hibernate;

public class Videogiochi extends Programmi {
    String categoria;

    public String getCategoria() {
        return categoria;
    }

    public void setCategoria(String categoria) {
        this.categoria = categoria;
    }
}
...
...
package it.ioprogrammo.librodb.hibernate;

public class OfficeSoftware extends Programmi {
    String licenza;

    public String getLicenza() {
        return licenza;
    }

    public void setLicenza(String licenza) {
        this.licenza = licenza;
    }
}
```

Le due entità che ci interessa mappare sono quindi delle classi figlie di *Programmi*, ma che comunque andranno ad essere inserite su due diverse tabelle. Quest'ultime avranno chiaramente delle colonne differenti, ma dovremo avere l'accortezza di scegliere un nome uguale per quanto riguarda la colonna dell'identificativo, perché questo dovrà essere condiviso tra le due tabelle.

```
CREATE TABLE `libro`.`VIDEOGIOCHI` (
 `ID_PROG` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
 `CATEGORIA` VARCHAR(45) NOT NULL,
 `NOME` VARCHAR(45) NOT NULL,
 PRIMARY KEY (`ID_PROG`)
)
ENGINE = InnoDB;
CREATE TABLE `libro`.`OFFICESOFTWARE` (
 `ID_PROG` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
 `LICENZA` VARCHAR(45) NOT NULL,
 `NOME` VARCHAR(45) NOT NULL,
 PRIMARY KEY (`ID_PROG`)
)
ENGINE = InnoDB;
```

Ogni tabella ha una colonna *ID_PROG* che utilizza come chiave primaria, nel nostro caso si tratterà di una chiave primaria "condivisa", perché a livello logico non esisteranno due "Programmi" con lo stesso identificativo. Vediamo ora come istruire Hibernate questo nuovo mapping

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
                                                    DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
```

```
<class name="it.ioprogrammo.librodb.hibernate.Programmi"
                                        abstract="true">
    <id name="id" column="ID_PROG">
        <generator class="increment"/>
    </id>
    <property name="nome" column="NOME" type="java.lang.String"
                                                        />

    <union-subclass name="it.ioprogrammo.
            librodb.hibernate.Videogiochi" table="VIDEOGIOCHI">
        <property name="categoria" column="CATEGORIA"
                                  type="java.lang.String" />

    </union-subclass>
    <union-subclass name="it.ioprogrammo.
        librodb.hibernate.OfficeSoftware" table="OFFICESOFTWARE">
        <property name="licenza" column="LICENZA"
                                  type="java.lang.String" />
    </union-subclass>
</class>
</hibernate-mapping>
```

La classe padre, *Programmi*, è stata definita come *abstract*, infatti non viene definita nessuna tabella relativa. L'unica cosa che è relativa alla classe padre è la property *nome*, che troviamo in entrambe le tabelle, e l'id che viene associato alla colonna *ID_PROG*. Le sottoclassi di questa gerarchia vengono definite con il tag *<union-subclass>*, dove viene riportata anche la tabella che si occuperà di questa sottoclasse. Oltre a queste tre maniere per effettuare il mapping di una gerarchia, possiamo anche utilizzare altri metodi, che talvolta possono essere la risultante di una combinazione tra quelli che abbiamo visto qui. In ogni caso bisogna vedere come poter applicare questi metodi alla nostra applicazione o come quest'ultima deve essere cambiata per adottarne uno.

LA GESTIONE DELLA CACHE

Visto che quando utilizziamo framework come Hibernate stiamo comunque parlando di gestire dati in qualche maniera, un argomento che deve essere affrontato di sicuro è quello relativo alla cache. Le prestazioni sono importanti nel nostro lavoro, specialmente quando parliamo di accesso ai dati, quindi vale la pena spendere un po' di tempo a parlare dei diversi modi in cui Hibernate gestisce il caching. Esistono all'interno di questo framework diversi modi per abilitare la cache dei dati. Prima di tutto dobbiamo sapere che la classe *Session*, fondamentale per tutte le operazioni, gestisce una sua cache relativa alla singola transazione che andiamo ad effettuare. Praticamente gli oggetti salvati e caricati dalla Session, rimangono in questa cache fino a quando la transazione non viene terminata. Per renderci conto di questo comportamento possiamo vedere il seguente esempio:

```
package it.ioprogrammo.librodb.hibernate;

import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class TestFirstLevelCache {

    public static void main(String[] args) throws Exception {
        SessionFactory sessionFactory=new
                    Configuration().configure().buildSessionFactory();
        Session session=sessionFactory.getCurrentSession();
        session.beginTransaction();

        Videogiochi test=new Videogiochi();
        test.setCategoria("RUOLO");
        test.setNome("Magic The Gathering");
        System.out.println("Oggetto creato");
```

```
            session.save(test);
            System.out.println("Chiamato save sull'oggetto");
            Thread.currentThread().sleep(8000);
            session.getTransaction().commit();
            System.out.println("Chiamato commit");
    }

}
```

In questa classe abbiamo creato un nuovo oggetto, l'abbiamo salvato e successivamente abbiamo inviato la commit sulla transazione.
Se andiamo a vedere i log possiamo renderci conto di quello che accade:

Oggetto creato
Hibernate: select max(ids_.ID_PROG) from (select ID_PROG from OFFICESOFTWARE union select ID_PROG from VIDEOGIOCHI) ids_
Chiamato save sull'oggetto Hibernate: insert into VIDEOGIOCHI (NOME, CATEGORIA, ID_PROG) values (?, ?, ?)
Chiamato commit

Praticamente, quando chiamiamo il metodo *save()* di *Session*, Hibernate si preoccupa di avere tutte le info che gli servono per avere un oggetto che può essere memorizzato (in questo caso l'id visto che stiamo utilizzando una union-class). Solo successivamente, quando viene richiamato il metodo *commit()*, Hibernate inserisce effettivamente l'entry nel database. Per essere più sicuri potete anche vedere che durante la *sleep()* l'entry ancora non è inserita nel database (con MySQL potete vederlo utilizzando MySQL Query Browser). Questo comportamento non è un bug ma è una cosa voluta. Immaginiamo di aprire una sessione, creare un'oggetto, salvarlo, successivamente modificarlo ed infine chiudere la sessione. Hibernate, giustamente, risparmia i diversi statement SQL che dovrebbe inviare al database per ottimizzare le prestazioni del nostro program-

ma. Gli oggetti in questa cache possono essere comunque rimossi utilizzando i metodi *evict()*, *clear()* e *flush()* di *Session*. All'interno di Hibernate è presente anche *StatelessSession*, che è una versione modificata di Session in cui non è presente la cache di primo livello che abbiamo visto ora. Oltre a questo tipo di cache esiste anche quella che viene definita di secondo livello, che è possibile utilizzare a livello di cluster o differenti JVM. Per abilitare questo tipo di cache, dobbiamo scegliere una delle varie implementazioni disponibili all'interno di Hibernate oppure costruirne una noi. Quelle che possiamo utilizzare implementano l'interfaccia *org.hibernate.cache*. Cache e hanno caratteristiche diverse. Infatti diverse implementazioni sono degli adattamenti di altri progetti opensource, che hanno implementato l'interfaccia Cache. Qui di seguito viene riportata la lista di cache disponibili

- **org.hibernate.cache.EhCache**: Plugin di EhCache per Hibernate. *http://ehcache.sourceforge.net*
- **org.hibernate.cache.HashtableCache**: Un'implementazione "leggera", che dovrebbe essere utilizzata solo in ambito di sviluppo
- **org.hibernate.cache.OSCache**: Plugin di OSCache, software di Opensymphony, per Hibernate. *http://www.opensymphony.com/oscache*
- **org.hibernate.cache.SwarmCache**: Plugin di SwarmCache, progetto opensource disponibile su *http://swarmcache.sourceforge.net*
- **org.hibernate.cache.TreeCache**: La soluzione TreeCache di JBossCache
- **org.hibernate.cache.OptimisticTreeCache**: Altro plugin di TreeCache che implementa diverse funzionalità

Nella seguente immagine vengono riportata le differenti caratteristiche delle cache che possiamo utilizzare con Hibernate.
Il tipo di cache riguarda il modo in cui vengono memorizzati i dati, in memoria, su disco o in ambiente cluster. La *Query Cache* è un'altra tipologia di cache di cui parleremo in seguito, mentre nelle seguenti colon-

Cache	Tipo	Cluster safe	Query Cache	Read-only	Nonstrict read write	Read-write	Transactional
Hashtable	Memory		Yes	Yes	Yes	Yes	
EhCache	Memory, disk		Yes	Yes	Yes	Yes	
OSCache	Memory, disk		Yes	Yes	Yes	Yes	
SwarmCache	Clustered (ip multicast)	Cluster invalidation		Yes	Yes		
TreeCache	Clustered (ip multicast)	Replication	Yes (clock sync req.)	Yes			Yes

Figura 2.6: Caratteristiche delle cache di Hibernate

ne vengono riportati i quattro diversi livelli di accesso che le cache supportano. Questi livelli servono per capire che tipo di concorrenza possiamo avere sui dati

- **Read-only**: Questo livello viene utilizzato quando dobbiamo soltanto leggere i dati. Chiaramente è uno di quelli più performanti
- **Read-write**: In questo caso abbiamo bisogno di utilizzare gli oggetti sia in lettura che in scrittura
- **Strict read-write**: Quando non c'è un eccessivo bisogno di aggiornare i dati e questa operazione viene eseguita occasionalmente allora il livello che ci interessa è esattamente questo
- **Transactional**: Il supporto per un livello completamente transazionale viene dato dai provider che supportano questa strategia di concorrenza

Il discorso relativo alle cache è molto dipendente dalle operazioni che vengono effettuate nella nostra applicazione, quindi nella maggior parte dei casi dobbiamo provare diverse soluzioni prima di trovare quella più performante per la nostra applicazione. Vediamo ora come abilitare una cache di secondo livello per i nostri dati. La prima operazione da fare è quella di abilitare la cache, dicendo ad Hibernate nel file di configurazione che siamo interessati ad utilizzare una tra le implementazioni disponibili:

```
<property name="hibernate.cache.provider_class">
        org.hibernate.cache.OSCacheProvider
</property>

<property name="hibernate.cache.use_second_level_cache">
        true
</property>
```

Poi abilitiamo il caching direttamente nel file di mapping della risorsa che vogliamo abilitare

```
<cache usage="read-only"/>
```

A questo punto abbiamo detto ad Hibernate che vogliamo la cache di secondo livello, che abbiamo scelto l'implementazione di OSCache e che la gerarchia *Programmi* (dove abbiamo inserito il tag *cache*) verrà gestita con una cache read-only. In questo modo abbiamo abilitato il secondo tipo di cache, le prestazioni della vostra applicazione potrebbero cambiare molto con una strategia di caching giusta. Come ultimo argomento vediamo ora l'ultima tipologia di cache disponibile su Hibernate, la *Query Cache*. Quando vengono effettuate alcune query c'è la possibilità di effettuare il caching, ma non dei risultati, bensì delle associazioni tra query e identificativi degli oggetti. Anche in questo caso dobbiamo capire se per la nostra applicazione questo tipo di cache può essere di qualche giovamento, senza magari abilitare cose che poi non cambiano assolutamente le performance. Per quanto riguarda la cache ed altre modifiche alla configurazione che possono migliorare sensibilmente le performance della nostra applicazione vi rimando alla seguente pagina ufficiale della documentazione *http://www.hibernate.org/hib_docs/v3/reference/en/html/performance.html*, che potete anche trovare nella distribuzione di Hibernate che avete scaricato sul vostro pc, all'interno della sottodirectory *doc*.

TOOL

Un framework famoso e utilizzato come Hibernate ha spinto molti sviluppatori a realizzare dei tool che possono aiutarci. Vengono qui riportati alcuni tool che possiamo utilizzare insieme ad Hibernate

- **Hibernate Tools**: Un plugin per l'IDE Eclipse, una serie di task Ant e una console per il database che possono risultare utili – *http://tools.hibernate.org*
- **NbXdoclet**: Un plugin per l'IDE Netbeans, che ci permette di avere tutte le funzionalità base di Hibernate all'interno di questo IDE – *http://nbxdoclet.sourceforge.net*
- **AndroMDA**: Genera il codice a partire da diagrammi UML e Hibernate può essere integrato – *http://www.andromda.org*

IBATIS

iBatis è un framework per il data mapping realizzato da Clinton Begin e che ora viene gestito come progetto opensource da Apache (*http://ibatis.apache.org*). Rispetto ad altri framework, come Hibernate, iBatis presenta una soluzione che viene detta ibrida, tra l'ORM e il classico SQL. Utilizzando questo framework, infatti, vedremo come esso sia proprio una sorta di middleware tra le nostre applicazioni e i database, senza interferire troppo nei rispettivi domini. Il problema classico dei tool ORM è quello che i due mondi, l'OO e i database, sono alcune volte molto diversi.
Spesso succede che quando ci appoggiamo troppo ad un ORM dobbiamo stravolgere il database e viceversa. iBatis cerca di proporsi come una soluzione per diverse situazioni applicative, permettendo di cablare il codice SQL all'esterno della nostra applicazione ma preservando comunque una definizione diretta dei vari comandi SQL. In questo modo possiamo utilizzare il meglio di entrambi i mondi senza venire a compromessi. Inoltre possiamo dire che utilizzando questo progetto riusciamo a individuare i giusti limiti di responsabilità all'interno di un progetto. Utilizzando altri progetti dobbiamo occuparci al tempo stesso del dominio Object Oriented e della sua rappresentazione nel modello relazionale. In questo modo possiamo anche partire da uno schema già esistente e cercare di applicarlo al nostro progetto. Come altri tool dello stesso genere, iBatis basa tutto sulla sua configurazione che permette di definire prima di tutto il modo in cui dobbiamo collegarci al nostro database. Oltre a questo, come nel caso di Hibernate, ci permette di definire diversi tipi di mapping, ma in questo caso la definizione cambia il suo significato come avremo modo di vedere. La seguente immagine riassume bene il mondo di iBatis

Figura 3.1: Architettura di iBatis

Attraverso un input, che può essere un tipo primitivo, un oggetto o un hash, il motore di iBatis elabora la richiesta che gli facciamo, basandosi sulla propria configurazione di accesso e di mapping. iBatis è un framework che attualmente ha diverse implementazioni per diversi linguaggi: Java, Ruby, C#. Chiaramente in questo libro ci dedicheremo a quella relativa a Java, però il fatto di avere implementazioni simili in altri linguaggi è comunque un pregio da ricordare.

CONFIGURAZIONE

Come punto di partenza per ogni tool dobbiamo capire il modo in cui questo viene configurato. Riportiamo qui di seguito la configurazione per la prima prova che faremo con iBatis

```
<?xml version="1.0" encoding="UTF-8" ?>
    <!DOCTYPE sqlMapConfig PUBLIC "-//iBATIS.com//DTD SQL Map
                                                    Config 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-config-2.dtd">
```

```
<sqlMapConfig>
    <properties
    resource="it/ioprogrammo/librodb/ibatis/SqlMapConfig.properties" />
    <settings
        cacheModelsEnabled="true"
        enhancementEnabled="true"
        lazyLoadingEnabled="true"
        maxRequests="32"
        maxSessions="10"
        maxTransactions="5"
        useStatementNamespaces="false"
    />

    <typeAlias alias="utente"
                    type="it.ioprogrammo.librodb.ibatis.Utente"/>
    <transactionManager type="JDBC" >
        <dataSource type="SIMPLE">
            <property name="JDBC.Driver" value="${driver}"/>
            <property name="JDBC.ConnectionURL" value="${url}"/>
            <property name="JDBC.Username"
                                    value="${username}"/>
            <property name="JDBC.Password" value="${password}"/>
        </dataSource>
    </transactionManager>
    <sqlMap resource="it/ioprogrammo/librodb/ibatis/Utente.xml" />
</sqlMapConfig>
```

Analizziamo ora passo passo tutte le informazioni contenute in questo file di configurazione. Prima di tutto viene definito un file di properties, *SqlMapConfig.properties*, che verrà incluso ed utilizzato da iBatis. All'interno di questo file possiamo inserire ad esempio le informazioni JDBC per la connessione al database. Nel nostro caso il contenuto di questo file è il seguente:

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://127.0.0.1/libro
username=root
password=mysql
```

Nel file di configurazione ogni proprietà definita altrove tramite file di properties viene richiamata utilizzando la seguente sintassi:

```
${NOMEVALORE}
```

Passiamo ora al tag *settings*. All'interno sono definiti tutti i settaggi relativi all'istanza di iBatis che prenderà in pasto questa configurazione. Vediamo il significato di ognuno di questi valori

- **cacheModelsEnabled**: Come già abbiamo visto con Hibernate, anche iBatis permette di abilitare una sorta di cache per i dati gestiti. In seguito vedremo in dettaglio come utilizzare questa feature
- **enhancementEnabled**: Permette di utilizzare cglib, una libreria opensource, per ottimizzare le prestazioni in particolari situazioni
- **lazyLoadingEnabled**: Abilitando questo flag è possibile caricare determinati dati in maniera lazy, ovvero solo quando servono e non subito
- **maxRequests**: Richieste massime attive allo stesso momento
- **maxSessions**: Sessioni attive allo stesso momento
- **maxTransactions**: Transazioni attive allo stesso momento
- **useStatementNamespaces**: Serve per abilitare i namespace per richiamare le funzionalità di mapping

Andando avanti nel file di configurazione troviamo la seguente riga:

```
<typeAlias alias="utente" type="it.ioprogrammo.librodb.ibatis.Utente"/>
```

che definisce semplicemente un alias utilizzato all'interno del file, per non

dover utilizzare in diversi casi il nome completo della classe ma solo l'alias. Successivamente viene definito il modo in cui iBatis deve collegarsi al database, attraverso la definizione in questo caso di una semplice connessione JDBC della quale vengono riportati tutti i parametri. In altre situazioni è possibile utilizzare JTA per avere un container che gestisce le transazioni della nostra applicazione. Al posto della configurazione JDBC era possibile definire un datasource attraverso DBCP (*Jakarta Commons Database Connection Pool*) o tramite JNDI nel caso in cui magari vogliamo utilizzare un datasource definito in un container come un Application Server. Alla fine del file di configurazione abbiamo inserito un tag *sqlMap*:

```
<sqlMap resource="it/ioprogrammo/librodb/ibatis/Utente.xml" />
```

che definisce una delle risorse che utilizzeremo nel nostro programma. Questo file XML indicato è un file di mapping, però in questo caso, a differenza di Hibernate, mappa una serie di statement SQL con un determinato metodo che può venire richiamato. Il significato di questo mapping è simile a quello di Hibernate, ma ci troviamo sostanzialmente in una situazione differente. Infatti, mentre un tool di ORM come Hibernate cerca di creare un'associazione stretta tra classe e tabella, attraverso iBatis questa associazione la possiamo creare noi a livello di SQL. Per ogni operazione che possiamo richiamare dal nostro programma, ci è consentito associare una classe che mappa i risultati, ma a livello di SQL definiamo in dettaglio cosa deve essere fatto. Per capire meglio questo meccanismo vediamo prima di tutto la definizione della tabella che utilizzeremo in questo esempio:

```
CREATE TABLE `libro`.`utente` (
 `ID` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
 `NOME` VARCHAR(45) NOT NULL,
 `COGNOME` VARCHAR(45) NOT NULL,
 `USERNAME` VARCHAR(45) NOT NULL,
```

```
 `PASSWORD` VARCHAR(45) NOT NULL,
 `EMAIL` VARCHAR(45) NOT NULL,
 `VERIFIED` BOOLEAN NOT NULL,
 PRIMARY KEY (`ID`)
)
ENGINE = InnoDB;
```

Di seguito riportiamo il file *Utente.xml*, in cui abbiamo definito una serie di operazioni attraverso le quali possiamo effettuare le quattro classiche operazioni di inserimento: lettura, modifica e cancellazione.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
"http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlMap namespace="Utente">

    <select id="getUtente"
                    resultClass="it.ioprogrammo.librodb.ibatis.Utente">
        SELECT ID as id,
        NOME as nome,
        COGNOME as cognome,
        USERNAME as username,
        PASSWORD as password,
        EMAIL as email,
        VERIFIED as verified
        FROM UTENTE
        WHERE ID = #value#
    </select>

    <insert id="insertUtente"
                parameterClass="it.ioprogrammo.librodb.ibatis.Utente">
        INSERT INTO
```

```
        UTENTE (NOME, COGNOME,USERNAME, PASSWORD, EMAIL,
                                                    VERIFIED)
        VALUES (#nome#, #cognome#, #username#, #password#,
        #email#, #verified#)
        <selectKey
            keyProperty="id"
            resultClass="int">
            SELECT LAST_INSERT_ID() AS value
        </selectKey>
    </insert>

    <update id="updateUtente"
                parameterClass="it.ioprogrammo.librodb.ibatis.Utente">
        UPDATE UTENTE
        SET NOME = #nome#,
        COGNOME = #cognome#,
        USERNAME = #username#,
        PASSWORD = #password#,
        EMAIL = #email#,
        VERIFIED = #verified#
        WHERE ID = #id#
    </update>

    <delete id="deleteUtente"
                parameterClass="it.ioprogrammo.librodb.ibatis.Utente">
        DELETE UTENTE
        WHERE ID = #id#
    </delete>

</sqlMap>
```

All'interno di questo file vediamo quattro diversi tag che definiscono delle operazioni collegate alla classe Utente. Quest'ultima è il classico

JavaBean che gestisce tutte le informazioni che ci interessano:

```
package it.ioprogrammo.librodb.ibatis;

public class Utente {
	String nome;
	String cognome;
	String username;
	String password;
	String email;
	int id;
	boolean verified;
...
...
```

Le operazioni che sono definite nel file di mapping vengono prima di tutto suddivise per tipologia (insert/update/select/delete). Ad ogni operazione viene dato un id, che sarà richiamato lato Java fornendo come parametro il JavaBean definito nell'attributo *parameterClass*. In questo modo i comandi SQL vengono generati a runtime, prendendo i vari parametri che vengono passati al motore di iBatis e costruendo dinamicamente le stringhe. All'interno dei vari metodi è possibile distinguere i parametri, che vengono definiti come di seguito:

```
USERNAME = #username#
```

iBatis, durante l'esecuzione, prenderà questi parametri dall'oggetto che gli viene fornito insieme alla chiamata. Chiaramente, il fatto che ci siano queste quattro tipologie di operazioni, con la possibile definizione di un identificativo, ci lascia aperta la possibilità di inserire diverse operazioni per una stessa tipologia. Ad esempio nel nostro programma potremmo aver bisogno di due diverse operazioni di delete, che possiamo semplicemente definire nel seguente modo con due diversi id

```
<delete id="deleteUtenteById"
                    parameterClass="it.ioprogrammo.librodb.ibatis.Utente">
     DELETE UTENTE
     WHERE ID = #id#
</delete>

<delete id="deleteUtenteByCredentials"
                    parameterClass="it.ioprogrammo.librodb.ibatis.Utente">
     DELETE UTENTE
     WHERE USERNAME = #username#
     AND PASSWORD = #password#
</delete>
```

C'è una cosa da approfondire per quanto riguarda l'operazione di insert, infatti, rispetto alle altre è presente un altro tag, *selectKey*

```
<selectKey
     keyProperty="id"
     resultClass="int">
     SELECT LAST_INSERT_ID() AS value
</selectKey>
```

Questo tag serve a valorizzare l'id della classe *Utente*. L'id, in questo caso, è stato definito con una colonna autoincrement di MySQL, in altri casi con Oracle e PostgreSQL potremmo definirli con una sequence, ma chiaramente non la definiremmo noi dal punto di vista del codice.
Proprio per questo nel momento in cui viene inserita la nuova entry nella tabella abbiamo comunque un oggetto *Utente*, nel quale non abbiamo valorizzato l'id. Attraverso questo tag riusciamo a valorizzare questa informazione, recuperandola dal database. In questo caso viene richiesto, attraverso un comando SQL specifico di MySQL, l'ultimo id inserito e valorizziamo la proprietà id dell'Utente con questo valore. Così facendo abbiamo dal punto di vista del codice un oggetto Utente

appena inserito, con il campo id valorizzato, senza dover procedere ad una successiva select per sapere quale sia questo valore (i programmatori che utilizzano JDBC sanno bene a cosa mi riferisco).

ESECUZIONE

Analizziamo ora come richiamare la configurazione che abbiamo scritto nel precedente paragrafo all'interno di un nostro programma. All'inizio dobbiamo richiamare la configurazione attraverso le classi della libreria di iBatis e poi vedremo come poter eseguire uno dei diversi comandi presenti:

```
package it.ioprogrammo.librodb.ibatis;

import java.io.IOException;
import java.io.Reader;
import java.sql.SQLException;

import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;

public class Insert {

    public static void main(String[] args) {
        String resource =
            "it/ioprogrammo/librodb/ibatis/iBatisSqlMapConfig.xml";
        Reader reader=null;
        try {
            reader = Resources.getResourceAsReader (resource);
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(-1);
```

```
        }
        SqlMapClient sqlMap =
                            SqlMapClientBuilder.buildSqlMapClient(reader);
        Utente utente=new Utente();
        utente.setNome("Federico");
        utente.setCognome("Paparoni");
        utente.setUsername("doc");
        utente.setEmail("doc@javastaff.com");
        utente.setPassword("NONLASO");
        utente.setVerified(true);
        try {
            sqlMap.insert("insertUtente",utente);
        } catch (SQLException e) {
            e.printStackTrace();
            System.exit(-1);
        }
        System.out.println("#ID "+utente.getId());
    }

}
```

Come prima operazione forniamo la nostra configurazione, memorizzata nel file *iBatisSqlMapConfig.xml*, a iBatis. Questo lo facciamo tramite un classico *java.io.Reader*, che viene richiamato per creare un oggetto *SqlMapClient*. Quest'ultimo verrà utilizzato per comunicare con il framework e avviare le operazioni desiderate. In seguito non facciamo altro che creare un oggetto *Utente*, valorizzare tutti i suoi campi (tranne id che verrà scelto in automatico dal db) e richiamare attraverso *SqlMapClient* l'operazione di insert classica. Subito dopo la insert avremo in output l'id dell'oggetto che abbiamo inserito, valore che viene recuperato dal database e settato nell'oggetto che passiamo come argomento. L'operazione di insert che noi abbiamo utilizzato aveva un id uguale a "*insertUtente*", quindi abbiamo passato questa stringa co-

me valore a *SqlMapClient*. Come abbiamo detto in precedenza possiamo inserire diverse operazioni della stessa tipologia, differenziandole proprio con l'id. Nell'attuale configurazione abbiamo inserito una select che prende un solo utente, ma potremmo essere interessati a caricare tutti gli utenti con una select differente. Per fare ciò non dobbiamo far altro che inserire questa nuova select, utilizzando chiaramente un id differente da quello già in uso:

```
<select id="getUtenti"
                    resultClass="it.ioprogrammo.librodb.ibatis.Utente">
    SELECT *
    FROM UTENTE
</select>
```

Ora che abbiamo aggiunto questa operazione al nostro file di configurazione, possiamo vedere come richiamare l'operazione di select, visualizzando in seguito i risultati:

```
package it.ioprogrammo.librodb.ibatis;

import java.io.IOException;
import java.io.Reader;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Iterator;

import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;

public class Select {

    public static void main(String[] args) {
```

```
            String resource =
                        "it/ioprogrammo/librodb/ibatis/iBatisSqlMapConfig.xml";
            Reader reader=null;
            try {
                  reader = Resources.getResourceAsReader (resource);
            } catch (IOException e) {
                  e.printStackTrace();
                  System.exit(-1);
            }
            SqlMapClient sqlMap =
                                    SqlMapClientBuilder.buildSqlMapClient(reader);
            ArrayList lista=null;
            try {
                  lista = (ArrayList)sqlMap.queryForList("getUtenti");
            } catch (SQLException e) {
                  e.printStackTrace();
                  System.exit(-1);
            }
            Iterator iterator=lista.iterator();
            Utente utente;
            while(iterator.hasNext()) {
                  utente=(Utente)iterator.next();
                  System.out.println(utente.getNome()+"
                                                        "+utente.getCognome());
            }
            System.out.println(lista.size());
      }

}
```

In questo caso la nostra operazione di select viene richiamata utilizzando il metodo *queryForList*. Il risultato della select, che dovrebbe essere una serie di oggetti *Utente* rappresentanti le entry del database, vengo-

no restituite all'interno di un oggetto *List*. Attraverso questo meccanismo di mapping, abbiamo all'interno del nostro oggetto *ArrayList* tutti gli utenti, che andiamo a visualizzare con un semplice *Iterator*.
Questo è uno dei diversi modi in cui possiamo mappare le informazioni che ci vengono restituite dal database. iBatis, per quanto riguarda il mapping, è estremamente flessibile, proprio perché essendo una soluzione ibrida tra l'ORM puro e l'SQL cerca di unire nel migliore dei modi le esigenze dal punto di vista della programmazione con quelle relative al mondo dei database.

SQL MAP

Il punto centrale di iBatis viene chiamato *SQL Map* non a caso. Infatti, come abbiamo già visto, viene proprio creato un mapping attraverso l'SQL definito per ogni operazione. Nell'esempio precedente abbiamo visto uno dei casi più semplici in cui ci possiamo trovare, una classe nel dominio della nostra applicazione che corrisponde ad una precisa tabella nel database. Ora vedremo un esempio più complesso, in cui iBatis riuscirà a risolvere il problema con una semplice configurazione. Immaginiamo di avere un database già definito, magari che noi possiamo soltanto utilizzare, in cui sono presenti due tabelle che interessano alla nostra applicazione. Per creare queste due tabelle possiamo avviare i seguenti script:

```
CREATE TABLE `libro`.`tabellaA` (
  `ID` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
  `INFO` VARCHAR(45) NOT NULL,
  `ALTRAINFO` VARCHAR(45) NOT NULL,
  PRIMARY KEY (`ID`)
)
ENGINE = InnoDB;

CREATE TABLE `libro`.`tabellaB` (
```

```
 `ID` INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
 `INFO` VARCHAR(45) NOT NULL,
 `ALTRAINFO` VARCHAR(45) NOT NULL,
 `ID_TABELLA_A` INTEGER UNSIGNED NOT NULL,
 PRIMARY KEY (`ID`),
 CONSTRAINT `FK_tabellaB_1` FOREIGN KEY `FK_tabellaB_1`
                                                    (`ID_TABELLA_A`)
   REFERENCES `tabellaa` (`ID`)
   ON DELETE CASCADE
   ON UPDATE CASCADE
)
ENGINE = InnoDB;
```

Dal punto di vista della nostra applicazione abbiamo una sola informazione da gestire poichè, per quanto riguarda la logica del nostro programma, le due tabelle sono gestite come una sola entità. In questo caso l'entità che utilizzeremo nel nostro programma è una semplice classe con alcuni valori:

```
package it.ioprogrammo.librodb.ibatis;

public class Informazione {
      int id;
      String info;
      String info2;
      String info3;

      public int getId() {
            return id;
      }
      public void setId(int id) {
            this.id = id;
      }
```

```
	public String getInfo() {
		return info;
	}
	public void setInfo(String info) {
		this.info = info;
	}
	public String getInfo2() {
		return info2;
	}
	public void setInfo2(String info2) {
		this.info2 = info2;
	}
	public String getInfo3() {
		return info3;
	}
	public void setInfo3(String info3) {
		this.info3 = info3;
	}
}
```

Con il mapping che abbiamo visto ci sarebbero dei problemi, perché non viene subito in mente il modo in cui gestire questa situazione.
iBatis ci permette di definire una certa classe per restituire le informazioni ed oltre a questo possiamo anche decidere all'interno di questa classe quali valori ritornati dalla query devono essere settati. Insomma riusciamo a definire un mapping riga per riga, inviando al database un comando SQL che ci permette di definire una join tra le due tabelle ed elaborando il risultato, incapsulandolo in quello che ci fa più comodo.
Il file di mapping dove viene definito l'operazione che abbiamo descritto viene riportato qui di seguito:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL Map 2.0//EN"
```

```
"http://www.ibatis.com/dtd/sql-map-2.dtd">

<sqlMap namespace="Informazione">

    <resultMap id="informazione"
                    class="it.ioprogrammo.librodb.ibatis.Informazione">
        <result property="id" column="A.ID"/>
        <result property="info" column="A.INFO"/>
        <result property="info2" column="B.INFO" />
        <result property="info3" column="B.ALTRAINFO" />
    </resultMap>

    <select id="getInformazione" parameterClass="int"
                                            resultMap="informazione">
        SELECT A.ID, A.INFO,
        B.INFO, B.ALTRAINFO
        FROM tabellaA A, tabellaB B
        WHERE A.ID=B.ID_TABELLA_A
        AND A.ID = #value#
    </select>

</sqlMap>
```

Come potete vedere nell'operazione di select abbiamo fatto una join tra le due tabelle che ci interessano, selezionando le informazioni che dovremo gestire nella nostra applicazione. Abbiamo inoltre definito un *resultMap*, invece del *resultClass* che avevamo utilizzato precedentemente. Attraverso questo tag indichiamo a iBatis che i risultati della select dovranno essere mappati in un certo modo, definito nel tag *resultMap* che ha un id uguale a "*informazione*". Abbiamo optato per inserire le informazioni nella classe *it.ioprogrammo.librodb.ibatis.Informazione*, settando tutte le varie proprietà di questa classe con le varie colonne che vengono restituite dalla nostra select. Per testare questo mapping non

dobbiamo far altro che richiamare questa funzione, passando come parametro un id e aspettando un risultato della classe *Informazione*:

```
package it.ioprogrammo.librodb.ibatis;

import java.io.Reader;

import com.ibatis.common.resources.Resources;
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;

public class TestResultMap {

    public static void main(String[] args) throws Exception {
        String resource =
                "it/ioprogrammo/librodb/ibatis/iBatisSqlMapConfig.xml";
        Reader reader= Resources.getResourceAsReader (resource);
        SqlMapClient sqlMap =
                        SqlMapClientBuilder.buildSqlMapClient(reader);
        Informazione
 informazione=(Informazione)sqlMap.queryForObject("getInformazione",1)
                                                                      ;
        System.out.println(informazione.getId());
        System.out.println(informazione.getInfo());
        System.out.println(informazione.getInfo2());
        System.out.println(informazione.getInfo3());
    }

}
```

Il metodo non è l'unico che iBatis fornisce. Ad esempio, potremmo inserire le informazioni restituite dalla select all'interno di un oggetto *Hashmap*, dove i nomi delle colonne sarebbero gli id. Oltre a questo pos-

siamo utilizzare queste “mappe” anche in ingresso, ovvero definire un oggetto che viene passato come parametro e grazie al quale possiamo generare un comando SQL:

```
<insert id="insertQualcosa"
                parameterClass="it.ioprogrammo.librodb.ibatis.Qualcosa">
    insert into TESTTABLE(ID, INFO1, INFO2)
    values (#id#, #info1#, #info2#)
</insert>
```

L’oggetto che passiamo come parametro avrà diverse proprietà che vengono recuperate da iBatis e che ci permettono di generare il comando SQL di nostro interesse. Seguendo questa tecnica è possibile definire mapping multipli e altre finezze che permettono di lavorare solo sulla configurazione, senza dover introdurre classi non pertinenti con la nostra applicazione.

LA CACHE DI IBATIS

Anche iBatis offre un servizio di cache per migliorare le performance delle nostre applicazioni. La configurazione di questa cache prevede quattro differenti tipologie:

- **MEMORY**: I risultati vengono salvati in memoria
- **FIFO**: Anche questa tipologia prevede il salvataggio in memoria, adottando inoltre la tecnica FIFO (First In First Out), ovvero vengono rimossi dalla cache gli oggetti più vecchi
- **LRU**: Si basa sempre sulla memoria e utilizza la tecnica LRU (Least Recently Used), che elimina gli oggetti meno utilizzati
- **OSCACHE**: Questa è un progetto opensource che è possibile utilizzare come cache provider per la nostra applicazione iBatis

Ogni diversa tecnica che adottiamo ha dei particolari valori che possia-

mo settare. *MEMORY*, ad esempio, può avere una configurazione addizionale che riguarda il modo in cui la JVM deve gestire gli oggetti:

- **STRONG**: In questo modo gli oggetti vengono eliminati dalla cache solo quando scade un certo timeout che impostiamo a livello di configurazione
- **SOFT**: Questo valore per la cache MEMORY cerca di tenere gli oggetti in memoria, fino a quando si raggiungono certi limiti che richiedono di liberare un po' di memoria
- **WEAK**: L'approccio più lasco, che sicuramente non va a riempire la memoria ma che richiede più comandi SQL da richiamare ogni tanto

Le informazioni vengono mantenute in cache nel rispetto della tipologia che noi andiamo a definire. Oltre a questo c'è da dire che ci sono dei "trigger" che permettono di capire quando bisogna svuotare la cache. Questi vengono definiti attraverso i seguenti tag :

- **flushInterval**: Viene definito l'intervallo di tempo dopo il quale gli oggetti in cache possono essere svuotati
- **flushOnExecute**: Quando abbiamo dei dati che vogliamo tenere in cache, questi potrebbero essere invalidati da altre operazioni. Ad esempio se abbiamo in cache i risultati di una select, le altre operazioni potrebbero invalidare questi dati quindi è necessario invalidare la cache quando vengono eseguiti certi comandi e questo tag serve proprio per definire le diverse operazioni che devono azionare questo meccanismo

Ora che abbiamo visto le diverse configurazioni che possono essere utilizzate, è arrivato il momento di vedere un esempio funzionante con la cache abilitata. La cosa principale che dobbiamo fare è quella di decidere quale tipologia di cache utilizzare e come configurarla. In questo esempio andremo a inserire nel mapping *Utente.xml* una cache per

l'operazione che seleziona tutti gli utenti:

```
<cacheModel id="cacheSelectUtenti" type="MEMORY">
    <flushOnExecute statement="insertUtente"/>
    <flushOnExecute statement="updateUtente"/>
    <flushOnExecute statement="deleteUtente"/>
    <property name="reference-type" value="WEAK"/>
</cacheModel>

<select id="getUtenti"
    resultClass="it.ioprogrammo.librodb.ibatis.Utente"
    cacheModel="cacheSelectUtenti"
>
    SELECT *
    FROM UTENTE
</select>
```

Nell'operazione *getUtenti* abbiamo segnalato che utilizzeremo una cache di nome *cacheSelectUtenti*. La definizione di quest'ultima evidenzia il tipo, *MEMORY*, le tre diverse operazioni che la invalideranno e il modo in cui utilizzeremo i riferimenti agli oggetti, *WEAK*. Ora per testare il funzionamento di questa cache vedremo un programma che prima avvia l'operazione getUtenti, poi l'avvia una seconda volta dopo un piccolo sleep del *Thread*. Successivamente inseriremo un nuovo utente, richiamando quindi l'operazione insertUtente ed infine verranno ricaricati un'altra volta tutti gli utenti:

```
package it.ioprogrammo.librodb.ibatis;

import java.io.Reader;
import java.util.ArrayList;

import com.ibatis.common.resources.Resources;
```

```
import com.ibatis.sqlmap.client.SqlMapClient;
import com.ibatis.sqlmap.client.SqlMapClientBuilder;

public class SelectCache {

    public static void main(String[] args) throws Exception {
        String resource =
                "it/ioprogrammo/librodb/ibatis/iBatisSqlMapConfig.xml";
        Reader reader= Resources.getResourceAsReader (resource);
        SqlMapClient sqlMap =
                        SqlMapClientBuilder.buildSqlMapClient(reader);
        long start;
        long end;

        System.out.println("Prima select");
        start=System.currentTimeMillis();
        ArrayList lista = (ArrayList)sqlMap.queryForList("getUtenti");
        end=System.currentTimeMillis();
        System.out.println("Tempo impiegato:"+(end-start));
        Thread.currentThread().sleep(1000);

        System.out.println("Select dopo lo sleep");
        start=System.currentTimeMillis();
        lista = (ArrayList)sqlMap.queryForList("getUtenti");
        end=System.currentTimeMillis();
        System.out.println("Tempo impiegato:"+(end-start));

        Utente utente=new Utente();
        utente.setNome("Federico");
        utente.setCognome("Paparoni");
        utente.setUsername("doc");
        utente.setEmail("doc@javastaff.com");
        utente.setPassword("NONLASO");
```

```
            utente.setVerified(true);
            sqlMap.insert("insertUtente", utente);

            System.out.println("Select dopo insert");
            start=System.currentTimeMillis();
            lista = (ArrayList)sqlMap.queryForList("getUtenti");
            end=System.currentTimeMillis();
            System.out.println("Tempo impiegato:"+(end-start));
    }

}
```

Una volta che viene avviato questo programma possiamo vedere l'output che ci farà capire bene cosa succede

Prima select
Tempo impiegato:440
Select dopo lo sleep
Tempo impiegato:0
Select dopo insert
Tempo impiegato:50

Come da copione, la seconda select impiega un tempo infinitesimale per recuperare le informazioni, visto che queste sono già in memoria. Però dopo l'operazione di insert la cache viene svuotata e quindi viene impiegato comunque del tempo per restituire i valori corretti.

DYNAMIC SQL

Il fatto di avere i comandi SQL direttamente nel file di mapping porta sicuramente dei vantaggi nella costruzione della nostra applicazione. Dobbiamo però pensare anche al fatto che in alcune situazioni non sappiamo a priori il preciso comando che dovremo comunicare al databa-

se, perché magari vogliamo realizzare una sorta di dinamicità all'interno del nostro codice. Anche in questo iBatis ci viene in aiuto, permettendoci di definire all'interno delle operazioni una sorta di SQL dinamico, che viene costruito a runtime in base a determinate situazioni.

```
<select id="getUtentiDynamic"
    parameterClass="it.ioprogrammo.librodb.ibatis.RicercaUtente"
    resultClass="it.ioprogrammo.librodb.ibatis.Utente"
>
    SELECT *
    FROM UTENTE
    <dynamic prepend="WHERE ">
        <isNull property="email" removeFirstPrepend="true"
        prepend="AND">
            EMAIL IS NULL
        </isNull>
        <isNotNull property="email" prepend="AND">
            EMAIL = #email#
        </isNotNull>
        <isNull property="password" prepend="AND">
            PASSWORD IS NULL
        </isNull>
        <isNotNull property="password" prepend="AND">
            PASSWORD = #password#
        </isNotNull>
    </dynamic>
</select>
```

La select che abbiamo riportato utilizza il meccanismo appena detto. Praticamente, nella ricerca degli utenti noi vogliamo dinamicamente cambiare la query in base ad un parametro che passiamo ad iBatis.
Questo parametro è una semplice classe che in questo caso contiene due possibili parametri sui quali vogliamo costruire la nostra query dinami-

ca (anche se non ha molto senso dal punto di vista logico, utilizziamo questa classe e questa operazione solo per far capire il tema trattato basandoci su quello che abbiamo realizzato fino ad ora)

```
package it.ioprogrammo.librodb.ibatis;

public class RicercaUtente {
	String email;
	String password;

	public String getEmail() {
		return email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	public String getPassword() {
		return password;
	}
	public void setPassword(String password) {
		this.password = password;
	}
}
```

iBatis, quindi, opera dinamicamente all'interno del tag dynamic, dove vediamo vengono esaminati i due valori delle proprietà di *RicercaUtente* e in base al loro valore (nullo o non in questo caso) viene appesa alla nostra query una certa stringa. I tag che possono essere utilizzati all'interno di dynamic, per esaminare una certa espressione e quindi decidere se scrivere l'SQL corrispondente, sono riportati di seguito

- **isEmpty**: Verifica se una variabile è vuota
- **isNotEmpty**: Verifica se una variabile non è vuota

- **isNull**: Verifica se una variabile è nulla
- **isNotNull**: Verifica se una variabile non è nulla
- **isPropertyAvailable**: A runtime viene controllata la presenza di un parametro
- **isNotPropertyAvailable**: A runtime viene controllata l'assenza di un parametro
- **isEqual**: Controlla l'uguaglianza
- **isNotEqual**: Controlla che non ci sia uguaglianza
- **isGreaterThan**: Verifica se una variabile è più grande di una certa proprietà o di un certo valore
- **isGreaterEqual**: Uguale alla precedente, in più viene inclusa l'uguaglianza
- **isLessThan**: Verifica se una variabile è più piccola di una certa proprietà o di un certo valore
- **isLessEqual**: Uguale alla precedente, in più viene inclusa l'uguaglianza

Ogni tag poi avrà bisogno di una serie di parametri aggiuntivi che permettono ad iBatis di applicare quella determinata regola. Per esempi e approfondimenti sul tema vi rimando alla documentazione ufficiale di iBatis. Per concludere c'è solo da notare che quando vengono utilizzate una serie di queste regole, che magari vanno in AND o in OR, di solito c'è il bug con la classica programmazione JDBC che porta a costruire una stringa di questo tipo:

```
SELECT X,Y,Z FROM ABC WHERE AND X=5 AND Y=4;
```

Praticamente viene messo l'AND (o l'OR) anche quando non ce ne sarebbe bisogno. Proprio per questo motivo è possibile levare questa condizione dalla stringa SQL e inserirla nella configurazione del tag:

```
<isNull property="email" removeFirstPrepend="true"
        prepend="AND">
        EMAIL IS NULL
```

```
</isNull>
```

In questo tag abbiamo deciso che l'SQL andrà in AND con altre espressioni, utilizzando l'attributo *prepend*. Oltre a questo abbiamo indicato a iBatis di rimuovere il primo operatore logico, proprio per evitare di scrivere una comando SQL sbagliato. Chi ha visto codice JDBC (non ottimizato) che appunto costruiva dinamicamente query utilizzando operatori AND e OR sicuramente capirà quanto sia utile questo ulteriore meccanismo di iBatis.

ABATOR

Abator è un tool che ci permette di velocizzare il nostro lavoro utilizzando iBatis. Con questo strumento possiamo fare quanto segue:

- Generare automaticamente classi Java che aderiscono alla struttura delle nostre tabelle
- Generare automaticamente anche i file di mapping per iBatis, all'interno dei quali sono già definite alcune operazioni standard
- Creare uno strato DAO per la gestione degli oggetti Java generati

Abator è sicuramente un buon tool per iniziare a programmare con iBatis, anche perché l'unica cosa che dobbiamo fornire a questo tool è una configurazione, dove gli indichiamo come collegarsi al nostro database e cosa generare nel dettaglio

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE abatorConfiguration
  PUBLIC "-//Apache Software Foundation//DTD Abator for iBATIS
                                        Configuration 1.0//EN"
  "http://ibatis.apache.org/dtd/abator-config_1_0.dtd">

<abatorConfiguration>
```

```
<abatorContext id="LibroExample" generatorSet="Java2">
  <jdbcConnection driverClass="com.mysql.jdbc.Driver"
      connectionURL="jdbc:mysql://127.0.0.1/libro"
      userId="root"
      password="mysql">
   <classPathEntry location="c:/lib/mysql-connector-java-5.0.4-bin.jar" />
  </jdbcConnection>

  <javaTypeResolver >
    <property name="forceBigDecimals" value="false" />
  </javaTypeResolver>

  <javaModelGenerator targetPackage="test.model"
                                              targetProject="testsrc">
    <property name="enableSubPackages" value="true" />
    <property name="trimStrings" value="true" />
  </javaModelGenerator>

  <sqlMapGenerator targetPackage="test.xml"  targetProject="testsrc">
    <property name="enableSubPackages" value="true" />
  </sqlMapGenerator>

  <daoGenerator type="IBATIS"
                  targetPackage="it.ioprogrammo.librodb.ibatis.abator"
                                              targetProject="testsrc">
    <property name="enableSubPackages" value="true" />
  </daoGenerator>

  <table schema="LIBRO" tableName="UTENTE"
                                         domainObjectName="Utente">
    <property name="useActualColumnNames" value="true"/>
  </table>
</abatorContext>
```

```
</abatorConfiguration>
```

Dopo aver scritto questo file possiamo generare le classi, i file di configurazione e lo strato DAO richiamando Abator da linea di comando:

```
java –jar abator.jar abatorConfig.xml true
```

e avremo come risultato tutto quello che volevamo automaticamente.
Per maggiori informazioni potete consultare l'url
http://ibatis.apache.org/abator.html

ODBMS

Il classico database con il quale abbiamo avuto a che fare finora fa parte della categoria dei DBMS (*Database Management System*) relazionali, ovvero dove i nostri dati vengono strutturati attraverso degli schema, tabelle, righe e quant'altro. Questo non è l'unico modo in cui può essere implementato un database, infatti quello relazionale è uno dei modelli che ha avuto più successo tra le diverse tipologie e per questo motivo la stragrande maggioranza dei database utilizzati sono stati sviluppati secondo questo paradigma. Tuttavia, quello che viene naturale dal punto di vista dello sviluppatore è cercare di mappare, nella maniera più indolore, lo strato informativo della propria applicazione nei database relazionali. Abbiamo visto che questo viene fatto attraverso dei tool come Hibernate e iBatis che ci permettono di astrarre la nostra applicazione dal database e continuare a ragionare in un'ottica Object Oriented. Bisogna comunque dire che esistono anche dei database che supportano nativamente il paradigma OO e che per questo motivo vengono chiamati ODBMS (Object DBMS). Attualmente, nell'ambito dei progetti opensource si possono segnalare i seguenti:

- db4o – *http://www.db4o.com*
- Perst – *https://perst.dev.java.net*

Esistono molte altre realtà opensource ma non sono aggiornate come questi due progetti che meritano davvero un po' di attenzione. In questo capitolo ci occuperemo di db4o, un database ad oggetti che è disponibile sia nella versione Java che .NET. Questo database, sviluppato dalla db4object Inc., è gratuitamente utilizzabile con licenza GPL o sotto la dOCL (*db4o Opensource Compatibility License*). Per essere utilizzato in ambito com-

merciale richiede una licenza. Guardando le prestazioni che sono state estrapolate da *PolePosition* (un benchmark di vari database disponibile su *http://www.polepos.org*), si vede subito come db4o abbiamo delle prestazioni davvero interessanti.
Per il dettaglio di tutte le sue funzionalità vi rimando al sito, dove è possibile trovare molta documentazione a riguardo.

PRIMO CONTATTO

Iniziamo a vedere come è possibile utilizzare db4o da Java.
Prima di tutto dobbiamo scaricare la versione Java ed includere nel nostro classpath i jar che troviamo all'interno della cartella *lib*. Per quanto riguarda le differenti versioni di java sono presenti 3 diversi jar:

- **db4o-VERSIONE-java1.1.jar**: Supporta Java 1.1, quindi è utilizzabile anche nei device mobili che supportano quelle API Java
- **db4o-VERSIONE-java1.2.jar**: Questa è la versione che può essere utilizzata se abbiamo a che fare con versioni di Java che vanno dalle 1.2 alle 1.4
- **db4o-VERSIONE-java5.jar**: Per Java 5

Immaginiamo ora di avere una nostra applicazione in Java che gestisce i giocatori di calcio. La classe con quale possiamo definire il giocatore potrebbe essere la seguente:

```
package it.ioprogrammo.librodb.db4o;

public class Giocatore {

   private String nome;
   private String cognome;
```

```
private Ruolo ruolo;
private int eta;
private double ingaggio;

public String getNome() {
    return nome;
}

public void setNome(String nome) {
    this.nome = nome;
}

public String getCognome() {
    return cognome;
}

public void setCognome(String cognome) {
    this.cognome = cognome;
}

public int getEta() {
    return eta;
}

public void setEta(int eta) {
    this.eta = eta;
}

public double getIngaggio() {
    return ingaggio;
}

public void setIngaggio(double ingaggio) {
```

```
        this.ingaggio = ingaggio;
    }

    public Ruolo getRuolo() {
        return ruolo;
    }

    public void setRuolo(Ruolo ruolo) {
        this.ruolo = ruolo;
    }

    @Override
    public String toString() {
        return nome+" "+cognome+" : "+ruolo.name();
    }

}
```

Giocatore è un classico oggetto Java, con diversi campi e i relativi metodi get/set. Oltre a questo abbiamo ridefinito il metodo *toString()* e abbiamo utilizzato un enumeration *Ruolo* che ci permette di definire il diverso ruolo che ha il giocatore:

```
package it.ioprogrammo.librodb.db4o;

public enum Ruolo {
    ATTACCANTE("attaccante"),
    CENTROCAMPISTA("centrocampista"),
    DIFENSORE("difensore"),
    PORTIERE("portiere");

    private String ruolo;
    Ruolo(String ruolo) {
```

```
        this.ruolo=ruolo;
    }
}
```

Vogliamo ora vedere come memorizzare e successivamente recuperare delle istanze della classe *Giocatore* con db4o.
Per fare questo possiamo accedere a db4o in locale, puntando direttamente al file che rappresenta il database oppure connettendosi da remoto a un server db4o. Per accedere in maniera locale non dobbiamo far altro che richiamare un determinato file e utilizzare le API di db4o per salvare l'oggetto:

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;

public class TestLocalDb {
    public static void main(String a[]) {
        Giocatore giocatore=null;
        ObjectContainer db=Db4o.openFile("db4o.db");
        try {
            giocatore=new Giocatore();
            giocatore.setNome("Francesco");
            giocatore.setCognome("Totti");
            giocatore.setEta(29);
            giocatore.setIngaggio(20000);
            giocatore.setRuolo(Ruolo.ATTACCANTE);
            db.set(giocatore);
            db.commit();
        }
        catch (Exception e) {
            System.out.println(e.toString());
```

```
        }
        finally {
            db.close();
        }
    }
}
```

Attraverso il metodo statico *openFile()* della classe *Db4o* abbiamo inizializzato l'oggetto *ObjectContainer* che ci permette di dialogare con il database. Dopo di ciò non abbiamo fatto altro che settare tutti i campi del nostro oggetto Giocatore, salvarlo attraverso il metodo *set()* ed effettuare la commit con il relativo metodo *commit()*. Vediamo ora come leggere gli oggetti della classe *Giocatore* che abbiamo memorizzato:

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

public class TestLocalDbRead {
    public static void main(String a[]) {
        Giocatore giocatore=new Giocatore();
        ObjectContainer db=Db4o.openFile("db4o.db");
        ObjectSet result=db.get(giocatore);
        System.out.println("Giocatori trovati: "+result.size());
        while(result.hasNext()) {
            System.out.println(result.next());
        }
        db.close();
    }
}
```

In questo caso ci siamo sempre collegati in locale sul file di db4o e abbiamo utilizzato il metodo *get()*, passando come argomento un oggetto *Giocatore* non valorizzato che ci permette di avere tutti gli oggetti della classe *Giocatore* che sono memorizzati nel database. In un paragrafo successivo vedremo come poter effettuare delle query utilizzando tre diversi metodologie che vengono offerte da db4o. Ora per completare questo "primo contatto" vedremo anche come collegarsi al nostro database attraverso la rete. L'approccio di collegarsi ad un database tramite filesystem chiaramente ha dei limiti, specialmente quando vogliamo offrire un servizio anche a processi remoti. Proprio per questo c'è la possibilità di gestire un vero e proprio server su db4o, settando le credenziali per l'accesso come in un tradizionale database.

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectServer;

public class TestServer {

    public static void main(String a[]) {
        String USERNAME="doc";
        String PASSWORD="cod";
        int PORTA=30330;
        boolean running=true;
        ObjectServer server=Db4o.openServer("db4o.db",PORTA);
        server.grantAccess(USERNAME,PASSWORD);

        try {
            while(running){
```

```
                Thread.currentThread().sleep(20000);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

}
```

Il metodo *openServer()* di Db4o ci permette di avviare un server e di ottenere l'istanza di ObjectServer. In questo modo possiamo abilitare le credenziali per l'accesso. Il database è sempre un file, ma in questo caso viene data la possibilità di collegarsi da remoto. Nell'esempio precedente facciamo ciclare all'infinito il nostro programmino per lasciare il server in piedi, chiaramente è una soluzione abbastanza "artigianale" e quindi potrebbe essere gestita in una maniera migliore. Vediamo il codice del client che effettua il collegamento al database attraverso la rete:

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

public class TestClient {
    public static void main(String a[]) {
        String USERNAME="doc";
        String PASSWORD="cod";
        int PORTA=30330;
        ObjectContainer
```

```
        client=Db4o.openClient("localhost",PORTA,USERNAME,PASSWORD);
        ObjectSet result=client.get(new Giocatore());
        System.out.println("Giocatori trovati: "+result.size());
        while(result.hasNext()) {
            System.out.println(result.next());
        }
        client.close();
    }
}
```

Per effettuare il collegamento abbiamo passato come argomento al metodo *openClient()* di Db4o tutte le informazioni necessarie. Abbiamo quindi ottenuto un ObjectContainer, con il quale possiamo comportarci come quando ci colleghiamo in locale.

QUERY

Passiamo ora a vedere i diversi metodi che possiamo utilizzare con db4o per effettuare delle ricerche. I tre possibili modi in cui possiamo effettuare delle query sul nostro database vengono riportati di seguito:

- QBE (Query By Example)
- Native query
- SODA Query

Le Query By Example, già analizzate nel precedente paragrafo, passano al metodo *get()* di *ObjectContainer* un'istanza della classe che ci interessa. Il risultato sarà un *ObjectSet*, un'interfaccia presente nelle API di db4o che estende diverse interfacce delle API standard Java, come *Collection*, *Iterable*, *Iterator* e *List*.
All'interno di questo oggetto troveremo il risultato della nostra query. Per poter selezionare solo alcune istanze all'interno del

nostro database, ad esempio quelle che hanno un particolare valore per una loro variabile, non dobbiamo far altro che valorizzare questa variabile all'interno dell'oggetto che passiamo a ObjectContainer. Qui di seguito trovate un esempio di utilizzo di QBE:

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

public class QueryByExample {
    public static void main(String a[]) {
        Giocatore giocatore=new Giocatore();
        ObjectContainer db=Db4o.openFile("db4o.db");
        ObjectSet result=db.get(giocatore);
        System.out.println("Giocatori trovati: "+result.size());
        while(result.hasNext()) {
            System.out.println(result.next());
        }
        giocatore=new Giocatore();
        giocatore.setEta(29);
        result=db.get(giocatore);
        System.out.println("Giocatori di 29 anni trovati: "+result.size());
        while(result.hasNext()) {
            System.out.println(result.next());
        }
        db.close();
    }
}
```

Nella prima query vengono selezionate tutte le istanze di *Gioca-*

tore presenti nel nostro database. Con la seconda query, invece, avendo settato l'età del giocatore, riusciremo a recuperare tutte le istanze di *Giocatore* che hanno l'attributo *eta* settato a *29*. Chiaramente questo metodo per la ricerca è veloce ma non permette di avere un controllo più approfondito delle ricerche che dobbiamo fare. La seconda tipologia di query che db4o offre sono le Native Query. Con questo strumento possiamo effettuare delle query differenti da QBE, ottimizzate dove possibile da db4o. Attraverso il metodo *query()* di *ObjectContainer* possiamo utilizzare questa metodologia. Ma vediamo ora cosa è possibile realizzare:

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.Predicate;
import java.util.Iterator;
import java.util.List;

public class NativeQuery {
    public static void main(String a[]) {
        ObjectContainer db=Db4o.openFile("db4o.db");
        System.out.println("Selezione con Inner Predicate");
        List<Giocatore> giocatori = db.query(new Predicate<Giocatore>() {
            public boolean match(Giocatore giocatore) {
                return giocatore.getEta()==29;
            }
        });
        Iterator<Giocatore> iterator=giocatori.iterator();
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
```

```
        }

        //SELEZIONE CON PREDICATO PREDEFINITO
        System.out.println("Selezione con Predicate predefinito");
        giocatori = db.query(new RuoloPredicate());
        iterator=giocatori.iterator();
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        //SELEZIONE CON PREDICATO E COMPARATOR PREDEFINITO
        System.out.println("Selezione con Predicate e Comparator
                                                        predefiniti");
        giocatori = db.query(new RuoloPredicate(),new
                                                CognomeComparator());
        iterator=giocatori.iterator();
        while(iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        db.close();
    }
}
```

La prima ricerca che effettuiamo è quella dove viene definito il *Predicate* attraverso una inner class. *Predicate* è una classe astratta di db4o che noi possiamo utilizzare per definire quali tra gli oggetti di una determinata classe devono essere presi come risultato della nostra query. Questo è possibile implementando il metodo *match()*, dove dobbiamo ritornare *true* se l'oggetto soddisfa la nostra ricerca. La seconda ricerca che abbiamo effettuato nel precedente codice utilizza un *Predicate* che abbiamo definito:

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.query.Predicate;

public class RuoloPredicate extends Predicate<Giocatore>{

    public boolean match(Giocatore giocatore) {
        String ruolo;
        ruolo=giocatore.getRuolo().toString();
        if (ruolo.equals("ATTACCANTE"))
            return true;
        else
            return false;
    }

}
```

RuoloPredicate praticamente viene utilizzato per scegliere soltanto le istanze di *Giocatore* con il ruolo di attaccante. Vediamo infine l'ultima query, dove oltre al nostro *Predicate* abbiamo utilizzato un *CognomeComparator*, classe che implementa *QueryComparator* e serve per ordinare i risultati in base ad un nostro ordine particolare:

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.query.QueryComparator;

public class CognomeComparator implements
                                        QueryComparator<Giocatore> {

    public int compare(Giocatore arg0, Giocatore arg1) {
        String cognomeA=arg0.getCognome();
```

```
        String cognomeB=arg1.getCognome();
        int result = cognomeA.compareTo(cognomeB);
        return result;
    }

}
```

In questo caso *CognomeComparator* serve per ordinare i risultati in base al cognome. Come abbiamo visto, le Native Query sono uno strumento interessante per modellare le ricerche che dobbiamo effettuare su db4o. L'ultimo modo che possiamo utilizzare per fare delle query è *SODA* (*Simple Object Data Access*). Questa API ci permette di definire una serie di vincoli sulla query che dobbiamo effettuare, potendo anche inserire diversi AND o OR tra queste condizioni. Il suo livello di usabilità è inferiore rispetto alle Native Query perché, come vedremo il codice è leggermente meno organizzato/pulito.

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.query.Constraint;
import com.db4o.query.Query;

public class SODA {
    public static void main(String a[]) {
        ObjectContainer db=Db4o.openFile("db4o.db");

        System.out.println("Selezione di tutti gli oggetti");
        Query query=db.query();
        query.constrain(Giocatore.class);
```

```
        ObjectSet result=query.execute();
        while(result.hasNext()) {
            System.out.println(result.next());
        }

      System.out.println("Selezione di tutti gli oggetti con età uguale a 29");
        query=db.query();
        query.constrain(Giocatore.class);
        query.descend("eta").constrain(new Integer(29));
        result=query.execute();
        while(result.hasNext()) {
            System.out.println(result.next());
        }

        System.out.println("Selezione di tutti gli oggetti con età uguale a 29
                                        e nome uguale a Francesco");
        query=db.query();
        query.constrain(Giocatore.class);
        Constraint constr=query.descend("nome")
        .constrain("Francesco");
        query.descend("eta")
        .constrain(new Integer(29)).and(constr);
        result=query.execute();
        while(result.hasNext()) {
            System.out.println(result.next());
        }
    }
}
```

In questo caso le query vengono fatte utilizzando la classe *Query* di db4o. Questa classe ci permette di definire una serie di vincoli, modellati con la classe *Constraint*, che possiamo aggiungere a *Query*. Nel codice precedente abbiamo prima selezionato tut-

ti i giocatori, poi abbiamo selezionato quelli con età pari a *29*, aggiungendo un *Container*. Infine abbiamo aggiunto due diversi *Constraint* e li abbiamo messi in AND. Come API è leggermente meno usabile delle Native Query, anche se c'è da dire che permette di modellare dal punto di vista OO una ricerca sul database e soprattutto è più performante. Infatti db4o cerca di trasformare le Native Query in chiamate SODA proprio per questo motivo.

AGGIORNAMENTO E CANCELLAZIONE

Per completare la nostra panoramica sull'utilizzo classico che facciamo di un database, dobbiamo necessariamente vedere come poter aggiornare e/o cancellare un entry. Per quanto riguarda la modifica, in un database relazionale avremmo costruito un stringa SQL. In questo caso dobbiamo modificare l'oggetto che riusciamo a reperire dal database

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

public class Modifica {
    public static void main(String a[]) {
        ObjectContainer db=Db4o.openFile("db4o.db");
        Giocatore giocatore;
        giocatore=new Giocatore();
        giocatore.setNome("Francesco");
        giocatore.setCognome("Totti");
        ObjectSet result=db.get(giocatore);
```

```
        giocatore=(Giocatore)result.next();

        //MODIFICA
        giocatore.setIngaggio(30000);
        db.set(giocatore);
        db.close();
    }
}
```

Chiaramente la modifica può essere fatta su un oggetto che è stato recuperato nella sessione corrente, altrimenti il metodo *set()* inserirà un nuovo oggetto nel nostro database. La cancellazione, invece viene eseguita attraverso il metodo *delete()* di *ObjectContainer:*

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;

public class Cancellazione {
    public static void main(String a[]) {
        ObjectContainer db=Db4o.openFile("db4o.db");
        Giocatore giocatore;
        giocatore=new Giocatore();
        giocatore.setNome("Francesco");
        giocatore.setCognome("Totti");
        ObjectSet result=db.get(giocatore);
        giocatore=(Giocatore)result.next();

        //CANCELLAZIONE
        db.delete(giocatore);
```

```
        db.close();
    }
}
```

RELAZIONI

Parliamo ora di come poter collegare diverse tipologie di oggetti nel nostro database. Nel classico modello relazionale abbiamo la possibilità di avere le *Foreign Key* (FK), che ci permettono di gestire il collegamento tra diverse tabelle del nostro database. In db4o possiamo invece inserire degli oggetti come attributi del nostro oggetto ed avere salvati su database entrambi, appoggiandoci quindi alla classica relazione OO HAS-A. Cioè, se ho un oggetto *A*, che al suo interno gestisce un Vector di altri oggetti *B*, nel momento in cui vado a memorizzare l'oggetto *A*, avrò come risultato nel database un oggetto *A* e tutti gli oggetti *B* inseriti in *A*. Vediamo bene come funziona questa feature con un esempio. Prima di tutto iniziamo col definire una classe *Squadra* che gestisce un Vector di oggetti *Giocatore* che abbiamo precedentemente definito ed utilizzato:

```
package it.ioprogrammo.librodb.db4o;

import java.util.Vector;

public class Squadra {
    private String nome;
    private int punti;
    private int scudetti;
    private Vector giocatori;

    public String getNome() {
        return nome;
```

```
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public int getPunti() {
        return punti;
    }

    public void setPunti(int punti) {
        this.punti = punti;
    }

    public int getScudetti() {
        return scudetti;
    }

    public void setScudetti(int scudetti) {
        this.scudetti = scudetti;
    }

    public Vector getGiocatori() {
        return giocatori;
    }
    public void setGiocatori(Vector giocatori) {
        this.giocatori = giocatori;
    }

}
```

Per utilizzare questa classe, dobbiamo inizializzarla, settando i vari attributi ed inserendo un Vector con qualche giocatore. Suc-

cessivamente la inseriamo nel nostro database db4o come abbiamo fatto fino ad ora, ovvero utilizzando il metodo *set()* di *ObjectContainer*:

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import java.util.Vector;

public class InsertSquadra {
   public static void main(String a[]) {
      ObjectContainer db=Db4o.openFile("db4o.db");
      Giocatore giocatore1, giocatore2;
      giocatore1=new Giocatore();
      giocatore1.setNome("Francesco");
      giocatore1.setCognome("Totti");
      giocatore1.setEta(32);
      giocatore1.setIngaggio(20000);
      giocatore1.setRuolo(Ruolo.ATTACCANTE);

      giocatore2=new Giocatore();
      giocatore2.setNome("Amantino");
      giocatore2.setCognome("Mancini");
      giocatore2.setEta(28);
      giocatore2.setIngaggio(14000);
      giocatore2.setRuolo(Ruolo.CENTROCAMPISTA);

      Squadra squadra=new Squadra();
      squadra.setNome("Roma");
      squadra.setPunti(20);
      squadra.setScudetti(4);
```

```
        Vector giocatori=new Vector();
        giocatori.add(giocatore1);
        giocatori.add(giocatore2);
        squadra.setGiocatori(giocatori);

        db.set(squadra);
        db.close();
    }
}
```

A questo punto avremo sul nostro database tre diversi oggetti memorizzati: una squadra e due giocatori. Le istanze di *Giocatore* che abbiamo memorizzate sono visibili anche con la classica chiamata che ci permette di vedere tutte le istanze di una determinata classe. Quando invece andiamo a recuperare l'istanza di *Squadra*, avremo automaticamente settato il vettore con tutti i giocatori, come quando abbiamo inserito l'oggetto nel database.

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import java.util.Vector;
public class ReadSquadra {
    public static void main(String a[]) {
        Squadra squadra=new Squadra();
        ObjectContainer db=Db4o.openFile("db4o.db");
        ObjectSet result=db.get(squadra);
        System.out.println("Squadre trovate: "+result.size());
        Giocatore giocatore;
```

```
        while(result.hasNext()) {
            squadra=(Squadra)result.next();
            System.out.println("Nome: "+squadra.getNome());
            System.out.println("Punti: "+squadra.getPunti());
            System.out.println("Giocatori: ");
            Vector giocatori=squadra.getGiocatori();
            for(int i=0;i<giocatori.size();i++) {
                giocatore=(Giocatore)giocatori.elementAt(i);
                System.out.println(giocatore);
            }
        }
        db.close();
    }
}
```

Eseguendo questo semplice programma avremo il seguente output, che ci segnala la presenza di una squadra con due giocatori:

```
Squadre trovate: 1
Nome: Roma
Punti: 20
Giocatori:
Francesco Totti : ATTACCANTE
Amantino Mancini : CENTROCAMPISTA
```

Chiaramente, il fatto di dover mantenere una consistenza di tutti i dati riferiti dall'oggetto *Squadra*, può essere un problema. Immaginiamo di avere una catena di oggetti che vengono riferiti, nel momento in cui andiamo ad aggiornare qualcosa e a salvare, in teoria db4o dovrebbe andare a controllare tutti i cambiamenti che sono stati fatti e aggiornare di conseguenza.
Questo comportamento, se fosse settato per default rallente-

rebbe molto il nostro database. Proprio per questo motivo, per abilitare questo comportamento, ovvero avere un aggiornamento coerente per tutti gli oggetti che sono riferiti dall'oggetto che stiamo salvando, c'è una configurazione speciale da settare:

```
Db4o.configure().objectClass("it.ioprogrammo.librodb.Squadra").
                                                cascadeOnUpdate(true);
```

In questo modo, nel momento in cui andiamo a modificare uno degli oggetti *Giocatore* che la nostra istanza di *Squadra* gestisce, queste modifiche verranno riportate sul database. Lo stesso discorso vale per quanto riguarda la cancellazione, ovvero quando stiamo cancellando una squadra, dobbiamo abilitare la cancellazione in cascata che ci permette con una sola cancellazione di eliminare la squadra e tutti i giocatori ad essa associati:

```
Db4o.configure().objectClass("it.ioprogrammo.librodb.Squadra").
                                                cascadeOnDelete(true);
```


INFRASTRUTTURA CLIENT/SERVER

Abbiamo già visto come gestire la comunicazione client/server utilizzando le API di db4o. In questo paragrafo vedremo un'altra funzionalità che fornisce db4o per realizzare un'infrastruttura client/server. Utilizzando il metodo statico *openServer()* di Db4o è possibile mettere in piedi un vero e proprio server che può essere raggiunto dai client con le giuste credenziali per potersi collegare al database ed effettuare le classiche operazioni.
Oltre a questo è possibile aprire un'ulteriore porta di ascolto per il server, dove possono essere scambiati dei messaggi informativi. Possiamo fare un parallelismo con i protocolli noti, ad esempio l'FTP (*File Transfer Protocol*). In questo protocollo ab-

biamo una porta che serve per l'invio dei veri e propri dati, tipicamente la 20, e un'altra porta che serve per controllare la connessione, la 21. In questo modo i dati che devono passare sulla porta 20 vengono "orchestrati" dai comandi che passano sulla porta 21. In questo caso possiamo dire che il funzionamento è simile, praticamente abbiamo la connessione classica che serve per effettuare le operazioni sul database. Esiste inoltre un'altra connessione che possiamo stabilire con il processo server per inviargli dei messaggi informativi, che possiamo usare ad esempio per avviare processi lato server come un backup dei dati. Vediamo prima di tutto di modellare il comando che vogliamo inviare al server con una semplice classe:

```
package it.ioprogrammo.librodb.db4o;

public class Comando {
    private String action;
    private String sender;

    public String getAction() {
        return action;
    }

    public void setAction(String action) {
        this.action = action;
    }
    public String getSender() {
        return sender;
    }

    public void setSender(String sender) {
        this.sender = sender;
    }
```

```
}
```

Passiamo ora al server. In questo caso, per abilitare questo canale di comunicazione supplementare, dobbiamo settare tramite le API di db4o un oggetto come *MessageRecipient*, un'interfaccia di db4o che definisce un solo metodo, *processMessage()*, che viene richiamato per ogni messaggio che arriva sul server.
All'interno di questo esempio, per semplicità, abbiamo messo nella stessa classe *server* e *MessageRecipient:*

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectServer;
import com.db4o.messaging.MessageRecipient;

public class ServerListener implements MessageRecipient{

   static ServerListener serverListener;

   public ServerListener() {
   }

   public static void main(String a[]) {
      serverListener=new ServerListener();
      String USERNAME="doc";
      String PASSWORD="cod";
      int PORTA=30330;
      boolean running=true;
      ObjectServer server=Db4o.openServer("db4o.db",PORTA);
      server.grantAccess(USERNAME,PASSWORD);
server.ext().configure().clientServer().setMessageRecipient(serverListener);
```

```
        try {
            while(running){
                Thread.currentThread().sleep(5000);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void processMessage(ObjectContainer arg0, Object arg1) {
        Comando comando=(Comando)arg1;
        System.out.println("Ricevuto comando");
        System.out.println("Fonte: "+comando.getSender());
        System.out.println("Comando: "+comando.getAction());
    }
}
```

Il server, in questo caso, ha dovuto inserire questa nuova feature utilizzando la configurazione client/server che si ottiene nel seguente modo:

```
server.ext().configure().clientServer()
```

Il metodo *processMessage()* scrive semplicemente il comando appena questo arriva al server. Vediamo ora come inviare il messaggio informativo dal client:

```
package it.ioprogrammo.librodb.db4o;

import com.db4o.Db4o;
import com.db4o.ObjectContainer;
import com.db4o.ObjectSet;
import com.db4o.messaging.MessageSender;
```

```
import java.net.InetAddress;
import java.net.UnknownHostException;

public class ClientMessaging
{
    public static void main(String a[]) throws UnknownHostException
    {
        String USERNAME="doc";
        String PASSWORD="cod";
        int PORTA=30330;
        Comando comando=new Comando();
        comando.setAction("BACKUP");
        comando.setSender(InetAddress.getLocalHost().getHostName());
        ObjectContainer
         client=Db4o.openClient("localhost",PORTA,USERNAME,PASSWORD);

        MessageSender messageSender =
                    client.ext().configure().clientServer().getMessageSender();
        messageSender.send(comando);

        client.close();
    }
}
```

Per inviare il messaggio abbiamo utilizzato la classe *MessageSender* e, una volta inizializzato l'oggetto *Comando*, lo abbiamo inviato utilizzando il metodo *send()*.
In questo modo possiamo creare una serie di comandi che possono essere inviati al server, senza che il client rimanga collegato.
Come citato nell'esempio, potremmo avviare un backup del nostro database inviando un certo comando e avviando successivamente lato server un thread di gestione per il backup.

OBJECTMANAGER

Per gestire db4o è possibile utilizzare un'applicazione interessante che è possibile scaricare direttamente dal sito ufficiale di db4o: *ObjectManager*. Si tratta praticamente di una semplice applicazione scritta in Java che all'inizio ci offre la possibilità di collegarci al nostro database attraverso file oppure passando attraverso un server
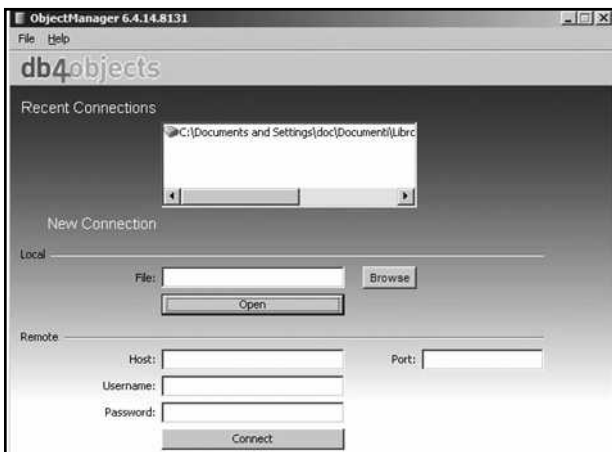
Figura 4.1: Avvio di ObjectManager

Una volta che ci siamo collegati possiamo controllare tutto il nostro database, vedendo tutti gli oggetti che sono memorizzati. Questi sono suddivisi per il nome della classe (comprensivo di package) e possono essere fatte delle ricerche attraverso un inputbox dove possiamo inserire una query.
Utilizzando questo tool è inoltre possible effettuare il backup del nostro database e avviare l'utility di deframmentazione.
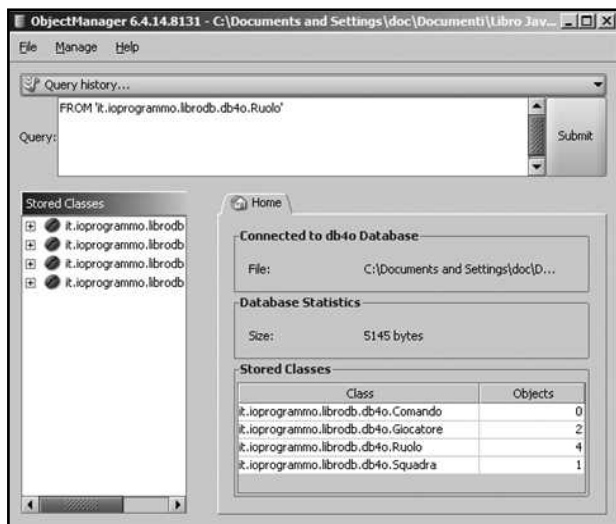
Figura 4.2: ObjectManager in esecuzione

REPLICA DEL DATABASE

Un'altra interessante feature che viene fornita da db4o è quella relativa alla replicazione del nostro database. Il fatto di avere due differenti database da mantenere aggiornati è uno scenario che si presenta talvolta nello sviluppo di applicazioni. Db4o mette a disposizione una semplice ma potente API che ci permette di replicare il nostro database con poche righe di codice.
La sincronizzazione del nostro database può avvenire in due modi differenti:

- Sincronizzazione con database db4o esterno
- Sincronizzazione con database relazionale

La prima modalità non fa altro che replicare il contenuto del nostro database db4o in un altro database dello stesso tipo.

La seconda modalità, invece, molto interessante, è quella relativa alla replicazione del db4o con un database relazionale.
Questa feature viene fornita utilizzando Hibernate, grazie al quale vengono mappati tutti gli oggetti in una tabella relazionale.
L'utilità della prima e soprattutto della seconda tipologia di sincronizzazione può essere trovata in molti scenari, ad esempio possiamo immaginare di avere due diversi livelli di accesso al database della nostra applicazione, differenziando tra un database di backend e uno di frontend.

ALTRE TECNOLOGIE

In questo libro non possiamo sicuramente sviscerare tutte le possibili tecnologie che è possibile utilizzare in Java per dialogare con il database. Proprio per questo motivo ora faremo un veloce riassunto di argomenti che non sono stati approfonditi ma che comunque meritano di essere visti e presi in considerazione quando affrontiamo decisioni riguardanti quale tecnologia utilizzare.

JDO

JDO (*Java Data Objects*) è una specifica venuta fuori dal Java Community Process (JCP), che cerca di definire una maniera standard per gestire le informazioni presenti sul database tramite Java. A rappresentare le tabelle del nostro database sono delle semplici classi Java, come ne abbiamo viste nel corso di questo libro, che non devono implementare nessuna interfaccia particolare. La connessione tra queste classi e database è rappresentata da file di configurazione XML che definiscono tutto.
Ci sono diverse versioni della specifica JDO, come riportato di seguito

- **JDO 1.0: JSR 12** – *http://www.jcp.org/en/jsr/detail?id=12*
- **JDO 2.0: JSR 243** - *http://www.jcp.org/en/jsr/detail?id=243*
- **JDO 2.1**: *http://db.apache.org/jdo*

Le idee che sono presenti in JDO sono abbastanza simili a quelle di vari altri framework, d'altronde la prima specifica pubblica di JDO risale al 2002. Esistono diverse implementazioni della API JDO che possiamo utilizzare:

- **JPOX**: Implementazione di riferimento per JDO – *http://www.jpox.org*
- **Kodo**: Supporta JDO 2.0 (versione solo commerciale) – *http://www.bea.com/kodo*
- **OJB**: ObjectRelationalBridge di Apache - *http://db.apache.org/ojb*
- **Speedo**: Implementazione JDO 1.0.1 – *http://speedo.objectweb.org*
- **TJDO**: TriActive JDO – *http://tjdo.sourceforge.net*

JPA

JPA (*Java Persistence API*) nasce come "costola" della JSR 220, definita per la versione 3.0 degli EJB. Anche questa è una specifica, come JDO, quindi può essere implementata o meno. Ma visto il suo collegamento con la nuova versione di EJB e soprattutto il fatto che sia stata definita proprio per creare una nuova API utilizzata in diversi ambiti per gestire le informazioni sul database, in breve tempo molti progetti opensource hanno incominciato a sviluppare un'implementazione di questa API. Il funzionamento è il solito, una classe POJO (Plain Old Java Object, ovvero quello che abbiamo sempre utilizzato nei vari progetti analizzati in questo libro) e una configurazione di mapping. L'interessante novità è che questa API si appoggia molto alle Annotations (introdotte in Java 5), permettendo quindi la definizione del mapping all'interno della stessa classe (che può essere visto come un bene o come un male, dipende dai punti di vista). Le implementazioni di questa API sono svariate:

- **Hibernate**: *http://www.hibernate.org*
- **TopLink Essentials**: *http://www.oracle.com/technology/products/ias/toplink/jpa/index.html*
- **Castor**: *http://www.castor.org*
- **JPOX**: *http://www.jpox.org*
- **GlassFish**:

https://glassfish.dev.java.net/downloads/persistence/JavaPersistence.html

- **OpenJPA**: *http://openjpa.apache.org*

In questo libro non è stato approfondito il mondo relativo agli Entity Bean, più che altro perché sono un argomento da affrontare in uno scenario differente ed approfondito. C'è da dire che JPA è il motore di persistenza di EJB3, quindi approfondire questa API può essere sicuramente buono perchè possiamo trovarci a lavorare con diverse librerie, su diversi fronti, ma comunque già siamo a conoscenza delle direttive fondamentali che vengono definite attraverso questa specifica.

CONCLUSIONI

Senza dilungarci troppo, dobbiamo sicuramente dire che esistono tanti progetti che possiamo utilizzare nel nostro programma Java per dialogare con un database. Non esiste di sicuro il "migliore", però quando dobbiamo decidere quale tecnologia utilizzare è giusto vedere anche le persone che seguono questo progetto. Ad esempio è poco furbo utilizzare dei progetti che sembrano abbandonati dai loro sviluppatori (ce ne sono tanti), anche perché se troviamo un problema all'interno di questo progetto dobbiamo rimboccarci le maniche e trovare da soli il bug. Questo non è l'unico metro di giudizio per un progetto opensource che possiamo utilizzare, ma è sicuramente uno dei suoi punti fondamentali. Altri progetti, che seppur validi, non hanno trovato spazio all'interno di questo libro sono i seguenti:

- **Cayenne**: *http://cayenne.apache.org*
- **Torque**: *http://db.apache.org/torque*